Parallel Program Analysis Framework for the DOE ACTS Toolkit

Allen D. Malony, Sameer Shende, Robert Ansell-Bell
{malony,sameer,bertie}@cs.uoregon.edu

Computer & Information Science Department
Computational Science Institute
University of Oregon

Talk Outline

- Project Goals and Challenges
- Computation Model for Performance Technology
- TAU Performance Framework
  - TAU architecture and performance system toolkit
- TAU Application Scenarios
  - Instrumentation examples
  - Object-oriented template libraries
  - Multi-level and asynchronous parallelism
  - Virtual machine execution
  - Hierarchical, hybrid parallel systems
- Future Plans and Conclusions

Performance Needs ➨ Performance Technology

- Observe/analyze/understand performance behavior
  - Multiple levels of software and hardware
  - Different types and detail of performance data
  - Alternative performance problem solving methods
  - Multiple targets of software and system application
- Robust AND ubiquitous performance technology
  - Broad scope of performance observability
  - Flexible and configurable mechanisms
  - Technology integration and extension
  - Cross-platform portability
  - Open layered and modular framework architecture

Complexity Challenges

- Computing system environment complexity
  - Observation integration and optimization
  - Access, accuracy, and granularity constraints
  - Diverse/specialized observation capabilities/technology
  - Restricted modes limit performance problem solving
- Sophisticated software development environments
  - Programming paradigms and performance models
  - Performance data mapping to software abstractions
  - Uniformity of performance abstraction across platforms
  - Rich observation capabilities and flexible configuration
  - Common performance problem solving methods

General Problem

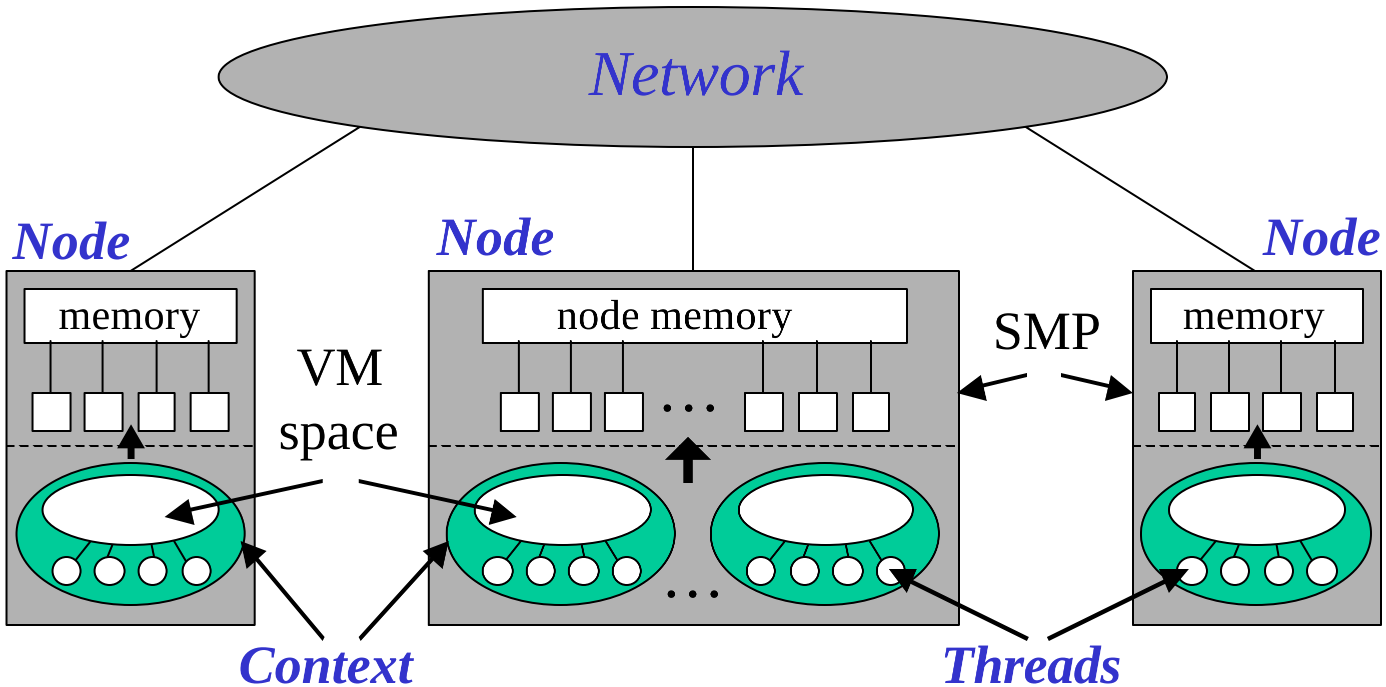
How do we create robust and ubiquitous performance technology for the analysis and tuning of parallel and distributed software and systems in the presence of (evolving) complexity challenges?

Computation Model for Performance Technology

- How to address dual performance technology goals?
  - Robust capabilities + widely available methodologies
  - Contend with problems of system diversity
  - Flexible tool composition/configuration/integration
- Approaches
  - Restrict computation types / performance problems
    - limited performance technology coverage
  - Base technology on abstract computation model
    - general architecture and software execution features
    - map features/methods to existing complex system types
    - develop capabilities that can adapt and be optimized

General Complex System Computation Model

- *Node*: physically distinct shared memory machine
  - Message passing *node interconnection network*
- *Context*: distinct virtual memory space within node
- *Thread*: execution threads (user/system) in context



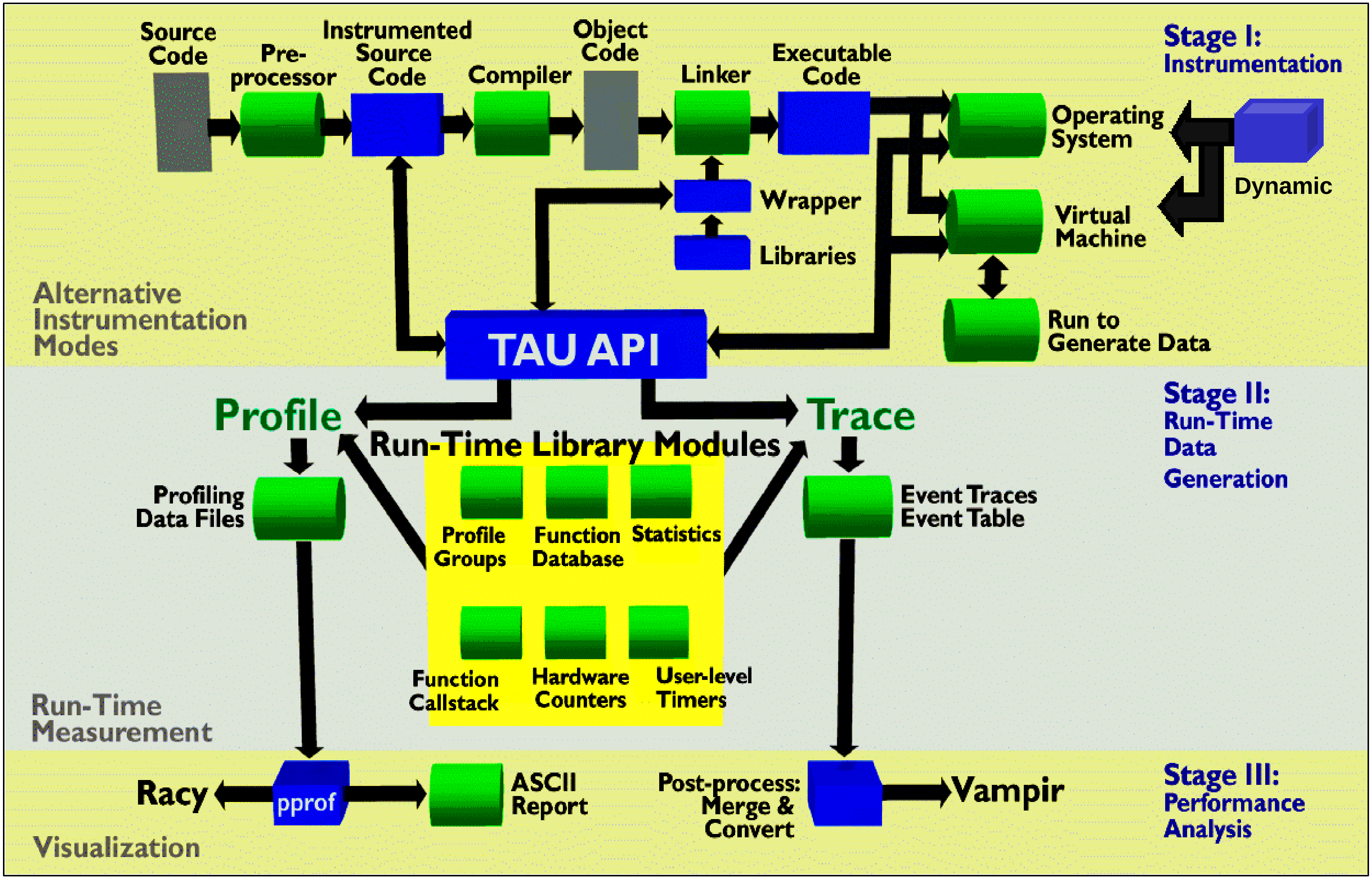
TAU Performance Framework

- Tuning and Analysis Utilities
- Performance system framework for scalable parallel and distributed high-performance computing
- Targets a general complex system computation model
  - *nodes* / *contexts* / *threads*
  - multi-level: system / software / parallelism
  - measurement and analysis abstraction
- Integrated toolkit for performance instrumentation, measurement, analysis, and visualization
  - portable performance profiling/tracing facility
  - open software approach

Targeted Research Areas

- Performance analysis for scalable parallel systems targeting multiple programming and system levels and the mapping between levels
- Program code analysis for multiple languages enabling development of new source-based tools
- Integration and interoperation support for building analysis tool frameworks and environments
- Runtime tool interaction for dynamic monitoring and adaptive applications

TAU Architecture



TAU Instrumentation

- Flexible, multiple instrumentation mechanisms
  - Source code
    - manual
    - automatic using PDT (tau_instrumentor)
  - Object code
    - pre-instrumented libraries (e.g., POOMA)
    - statically linked (e.g., MPI wrapper library)
    - dynamically linked (e.g., JVM profiling interface)
  - Executable code
    - dynamic instrumentation using DynInstAPI (tau_run)
  - Virtual machine

TAU Instrumentation (continued)

- Common target measurement interface (TAU API)
- C++ (object-based) design and implementation
  - Macro-based, using constructor/destructor techniques
  - Function, classes, and templates
  - Uniquely identify functions and templates
    - name and type signature (name registration)
    - static object creates performance entry
    - dynamic object receives static object pointer
    - runtime type identification for template instantiations
  - C and Fortran instrumentation variants
- Instrumentation and measurement optimization

TAU Measurement

- Performance information
  - High resolution timer library (real-time / virtual clocks)
  - Generalized software counter library
  - Hardware performance counters
    - PCL (Performance Counter Library) (ZAM, Germany)
    - PAPI (Performance API) (UTK, Ptools Consortium)
    - consistent, portable API
- Organization
  - Node, context, thread levels
  - Profile groups for collective events (runtime selective)
  - Mapping between software levels

TAU Measurement (continued)

- Profiling
  - Function-level, block-level, statement-level
  - Supports user-defined events
  - TAU profile (function) database (PD)
  - Function callstack
  - Hardware counts instead of time
- Tracing
  - Profile-level events
  - Interprocess communication events
  - Timestamp synchronization
- User-controlled configuration (configure)

What Data Can TAU Generate?

- Time spent exclusively and inclusively in each function.
- Number of times each function called.
- Number of profiled functions (subroutines) it called.
- Function mean time/call on all nodes/contexts/threads.
- Exclusive/inclusive time for each function invocation.
- Hardware counts: flops, instructions issued, cycles.
- Communication functions only or communication + I/O.
- Statement-level and block-level profiling.
- Profile statistics such as exclusive time standard deviation.

TAU Measurement API

- Configuration
  - TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(myNode);
    TAU_PROFILE_SET_CONTEXT(myContext);
    TAU_PROFILE_EXIT(message);
- Function and class methods
  - TAU_PROFILE(name, type, group);
- Template
  - TAU_TYPE_STRING(variable, type);
    TAU_PROFILE(name, type, group);
    CT(variable);
- User-defined timing
  - TAU_PROFILE_TIMER(timer, name, type, group);
    TAU_PROFILE_START(timer);
    TAU_PROFILE_STOP(timer);

TAU Measurement API (continued)

- User-defined events
  - TAU_REGISTER_EVENT(variable, event_name);
    TAU_EVENT(variable, value);
    TAU_PROFILE_STMT(statement);
- Mapping
  - TAU_MAPPING(statement, key);
    TAU_MAPPING_OBJECT(funcIdVar);
    TAU_MAPPING_LINK(funcIdVar, key);
  - TAU_MAPPING_PROFILE (FuncIdVar);
    TAU_MAPPING_PROFILE_TIMER(timer, FuncIdVar);
    TAU_MAPPING_PROFILE_START(timer);
    TAU_MAPPING_PROFILE_STOP(timer);
- Reporting
  - TAU_REPORT_STATISTICS();
    TAU_REPORT_THREAD_STATISTICS();

TAU Profile Groups (examples)

Profile Group	Description	Example
TAU_DEFAULT	All profile groups	--profile
TAU_MESSAGE	Message Class	--profile message
TAU_PETE	PETE	--profile pete+message
TAU_IO	IO functions	--profile io
TAU_FIELD	Field functions	--profile field+viz
TAU_LAYOUT	Field layout	--profile layout
TAU_MESHES	Meshes	--profile sub+meshes
TAU_PARTICLE	Particle	--profile io+particle
TAU_USER	User defined	--profile user

Timing of Multi-threaded Applications

- Capture timing information on per thread basis
- Two alternative
  - Wall clock time
    - works on all systems
    - user-level measurement
  - OS-maintained CPU time (e.g., Solaris, Linux)
    - thread virtual time measurement
- TAU supports both alternatives
  - CPUTIME module profiles user+system time
- PAPI thread timing

TAU Analysis

- Profile analysis
  - *Pprof*
    - parallel profiler with texted based display
  - *Racy*
    - graphical interface to pprof
- Trace analysis
  - Trace merging and clock adjustment (if necessary)
  - Trace format conversion (ALOG, SDDF, PV, Vampir)
  - *Vampir* (Pallas)

Using PPROF

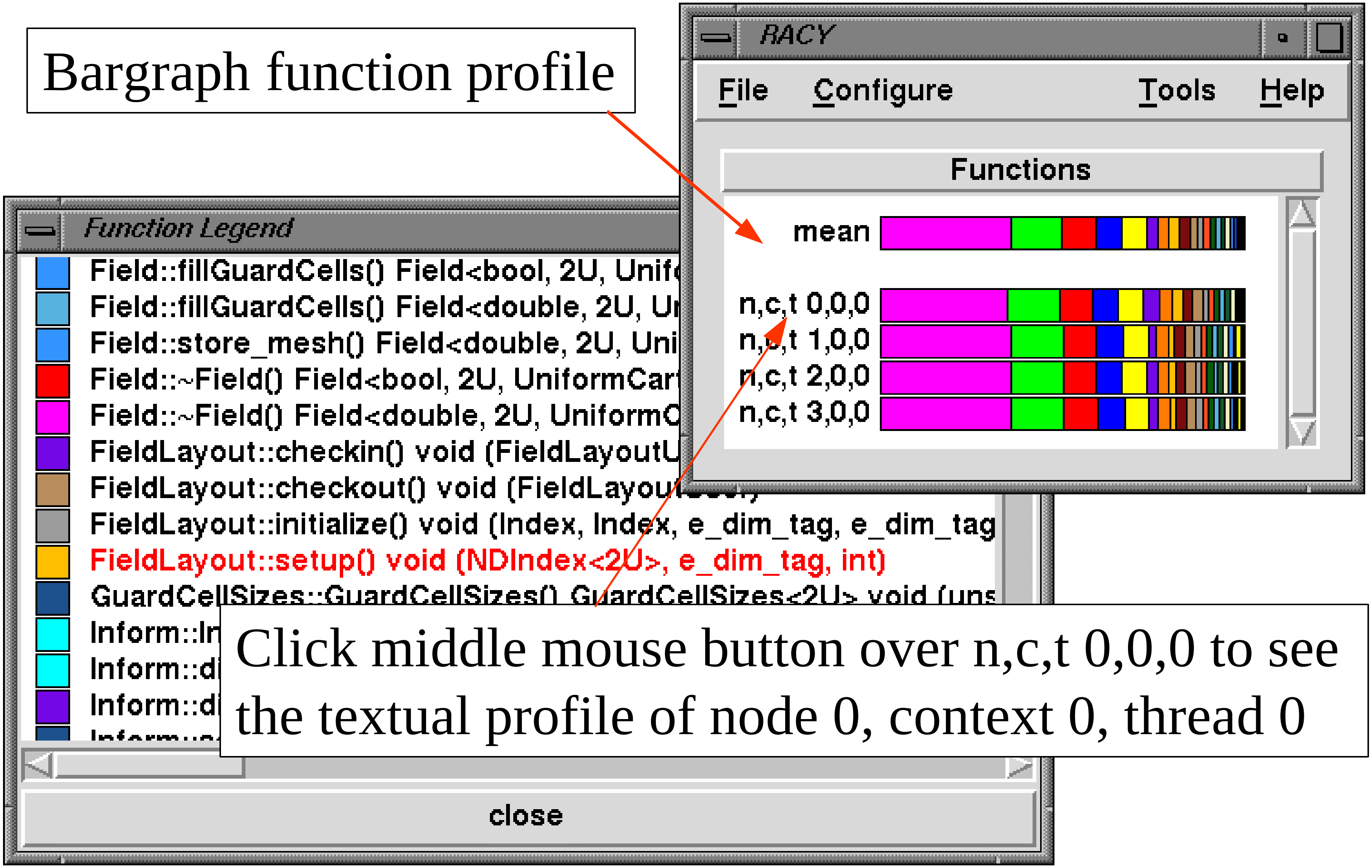
File Edit Apps Options Buffers Tools Comint1 Comint2 History Help

Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

```
[strawberry] /users/sameer/pp/conejo1 % pprof
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0       23,145    36:36.980           1       14378 2196980000 main int(int,char*[])
 12.1     4:26.306     4:26.306       45792           0       5816 apply BrickExpression<3U, L
 11.2          234     4:06.693         100         196    2466930 assign(IndexedBareField) PE
 10.9     3:59.772     3:59.772          96           0    2497625 apply BrickExpression<3U, L
  7.7          576     2:49.948        3705       23775      45870 Field::fillGuardCells TecMa
--**-XEmacs: *shell*      (Shell: run)---- 1%-------------------------------------------
usage: pprof [-c|-b|-m|-t|-e|-i|-v] [-r] [-s] [-n num] [-f filename] [-l] [node numbers]
 -c : Sort according to number of Calls
 -b : Sort according to number of suBroutines called by a function
 -m : Sort according to Milliseconds (exclusive time total)
 -t : Sort according to Total milliseconds (inclusive time total)  (default)
 -e : Sort according to Exclusive time per call (msec/call)
 -i : Sort according to Inclusive time per call (total msec/call)
 -v : Sort according to Standard Deviation (excl usec)
 -r : Reverse sorting order
 -s : print only Summary profile information
 -n <num> : print only first <num> number of functions
 -f filename : specify full path and Filename without node ids
 -l : List all functions and exit
 [node numbers] : prints only info about all contexts/threads of given node numbers
--**-XEmacs: *shell*      (Shell: run)----99%-------------------------------------------
```

Using RACY

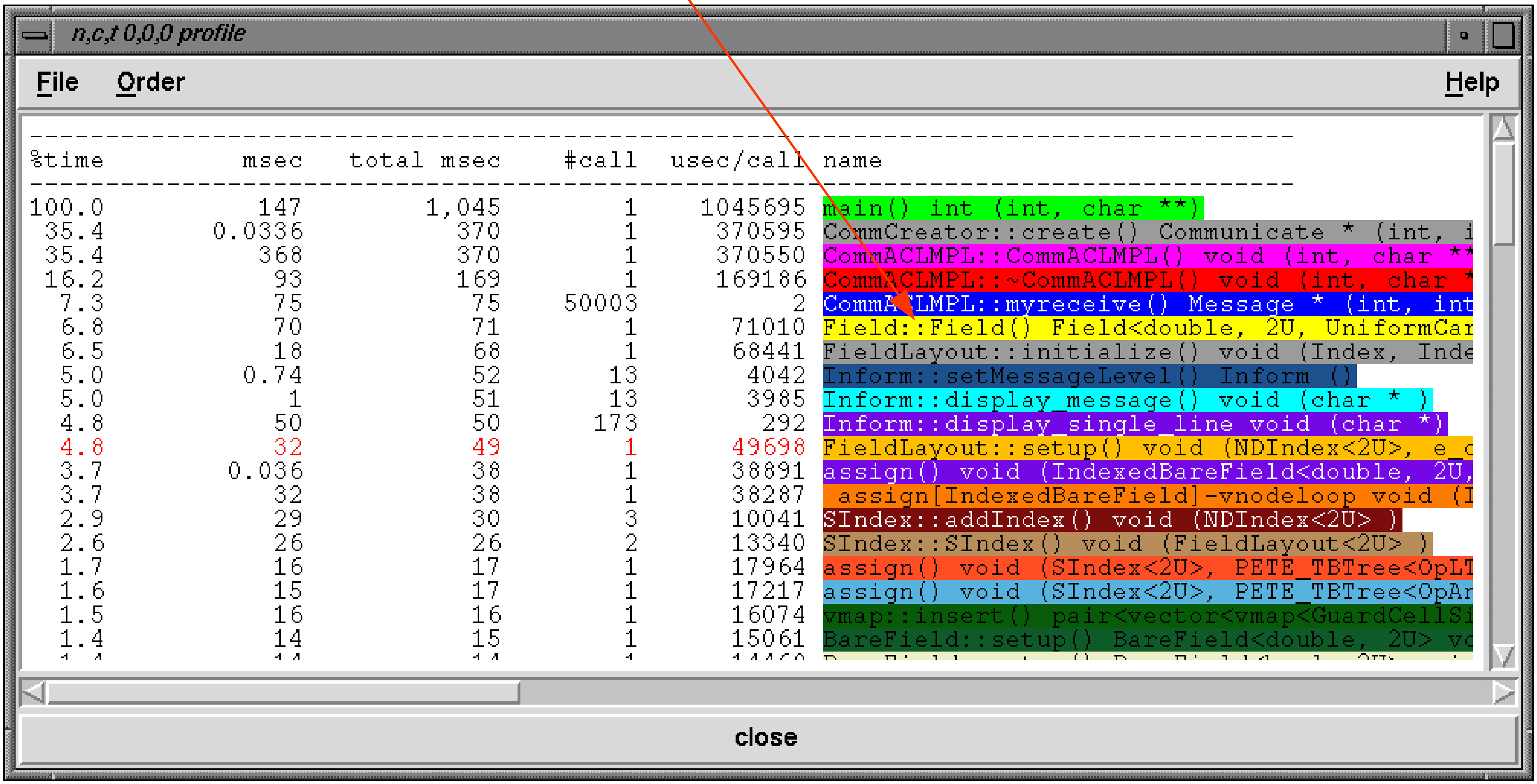
Bargraph function profile



Click middle mouse button over n,c,t 0,0,0 to see the textual profile of node 0, context 0, thread 0

Using RACY (continued)

Click the third mouse button over a function to highlight it in all racy windows.



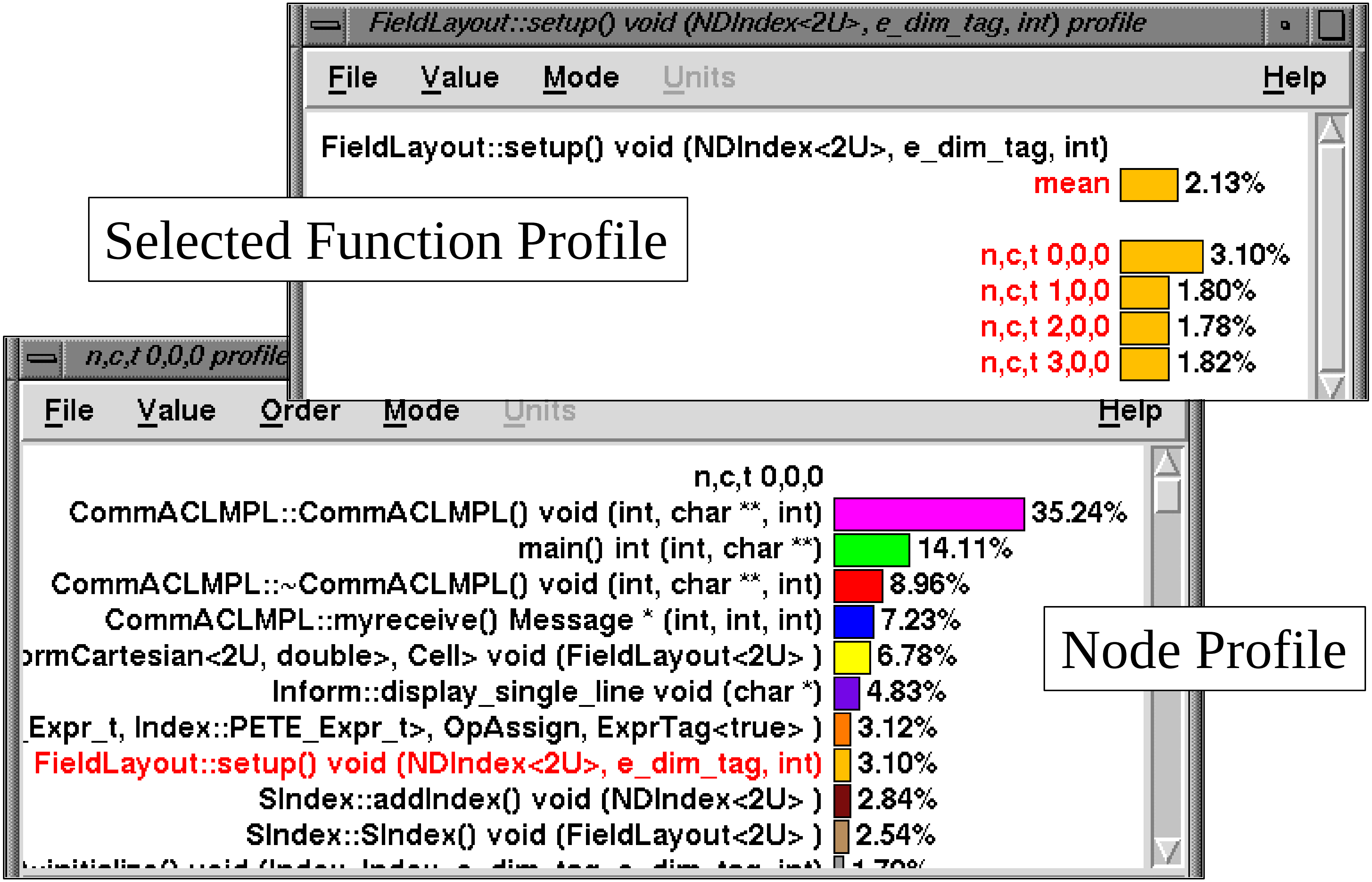
n,c,t 0,0,0 profile

File Order Help

%time	msec	total msec	#call	usec/call	name
100.0	147	1,045	1	1045695	main() int (int, char **)
35.4	0.0336	370	1	370595	CommCreator::create() Communicate * (int, i
35.4	368	370	1	370550	CommACLMPL::CommACLMPL() void (int, char **
16.2	93	169	1	169186	CommACLMPL::~CommACLMPL() void (int, char *
7.3	75	75	50003	2	CommACLMPL::myreceive() Message * (int, int
6.8	70	71	1	71010	Field::Field() Field<double, 2U, UniformCar
6.5	18	68	1	68441	FieldLayout::initialize() void (Index, Inde
5.0	0.74	52	13	4042	Inform::setMessageLevel() Inform ()
5.0	1	51	13	3985	Inform::display_message() void (char *)
4.8	50	50	173	292	Inform::display_single_line void (char *)
4.8	32	49	1	49698	FieldLayout::setup() void (NDIndex<2U>, e_c
3.7	0.036	38	1	38891	assign() void (IndexedBareField<double, 2U,
3.7	32	38	1	38287	assign[IndexedBareField]-vnodeloop void (I
2.9	29	30	3	10041	SIndex::addIndex() void (NDIndex<2U>)
2.6	26	26	2	13340	SIndex::SIndex() void (FieldLayout<2U>)
1.7	16	17	1	17964	assign() void (SIndex<2U>, PETE_TBTree<OpLT
1.6	15	17	1	17217	assign() void (SIndex<2U>, PETE_TBTree<OpAr
1.5	16	16	1	16074	vmap::insert() pair<vector<vmap<GuardCellSi
1.4	14	15	1	15061	BareField::setup() BareField<double, 2U> vo

close

Using RACY (continued)



TAU Status

- Usage (selective)
  - Platforms
    - IBM SP, SGI Origin 2K, Intel Teraflop, Cray T3E, HP, Sun, Windows 95/98/NT, Alpha/Pentium Linux cluster, IA-64
  - Languages
    - C, C++, Fortran 77/90, HPF, pC++, HPC++, Java, OpenMP
  - Communication libraries
    - MPI, PVM, Nexus, Tulip, ACLMPL
  - Thread libraries
    - pthreads, Tulip, SMARTS, Java,Windows
  - Compilers
    - KAI, PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray, IBM

TAU Status (continued)

- Application libraries
  - Blitz++, A++/P++, ACLVIS, PAWS
- Application frameworks
  - POOMA, POOMA-2, MC++, Conejo, PaRP
- Other projects
  - ACPC, University of Vienna: Opus/HPF
  - KAI and Pallas: OpenMP/MPI

- TAU profiling and tracing toolkit (Version 2.8)
- Extensive 70-page *TAU User's Guide*
- *http://www.acl.lanl.gov/tau*
- *http://www.cs.uoregon.edu/research/paracomp/tau*

TAU Application Scenarios

- Instrumentation examples
  - Instrumentation of C++ source and templates
  - Instrumentation of multi-threaded code
- Object-oriented (C++) template libraries
  - Template-derived code performance measurement
  - Array classes and expression transformation
  - Source code performance mapping
- Multi-level and asynchronous computation
  - Multi-threaded parallel execution
  - Asynchronous runtime system scheduling
  - Parallel performance mapping

TAU Application Scenarios (continued)

- Hardware performance measurement
  - Integration of external performance technology
  - Cross-platform hardware counter API
- Virtual machine execution
  - Abstract thread-based performance measurement
  - Performance measurement integration in virtual machine
- Hierarchical, hybrid (mixed model) parallel systems
  - Portable shared memory and message passing APIs
  - Combined task and data parallel execution
  - Performance system configuration and model mapping

C++ Instrumentation Using TAU API

```
int main(int argc, char **argv)
{
 TAU_PROFILE("main()", "int (int, char **)" TAU_DEFAULT);
 TAU_PROFILE_TIMER(ft, "For-loop-main", " ", TAU_USER);

 TAU_PROFILE_START(ft);
  for (int j = 0; j < N; j++)
   {
     cout <<"Something…"<<endl;
   }
  TAU_PROFILE_STOP(ft);
} // routines & methods need just one TAU_PROFILE
```

Instrumentation of C++ Templates Using TAU API

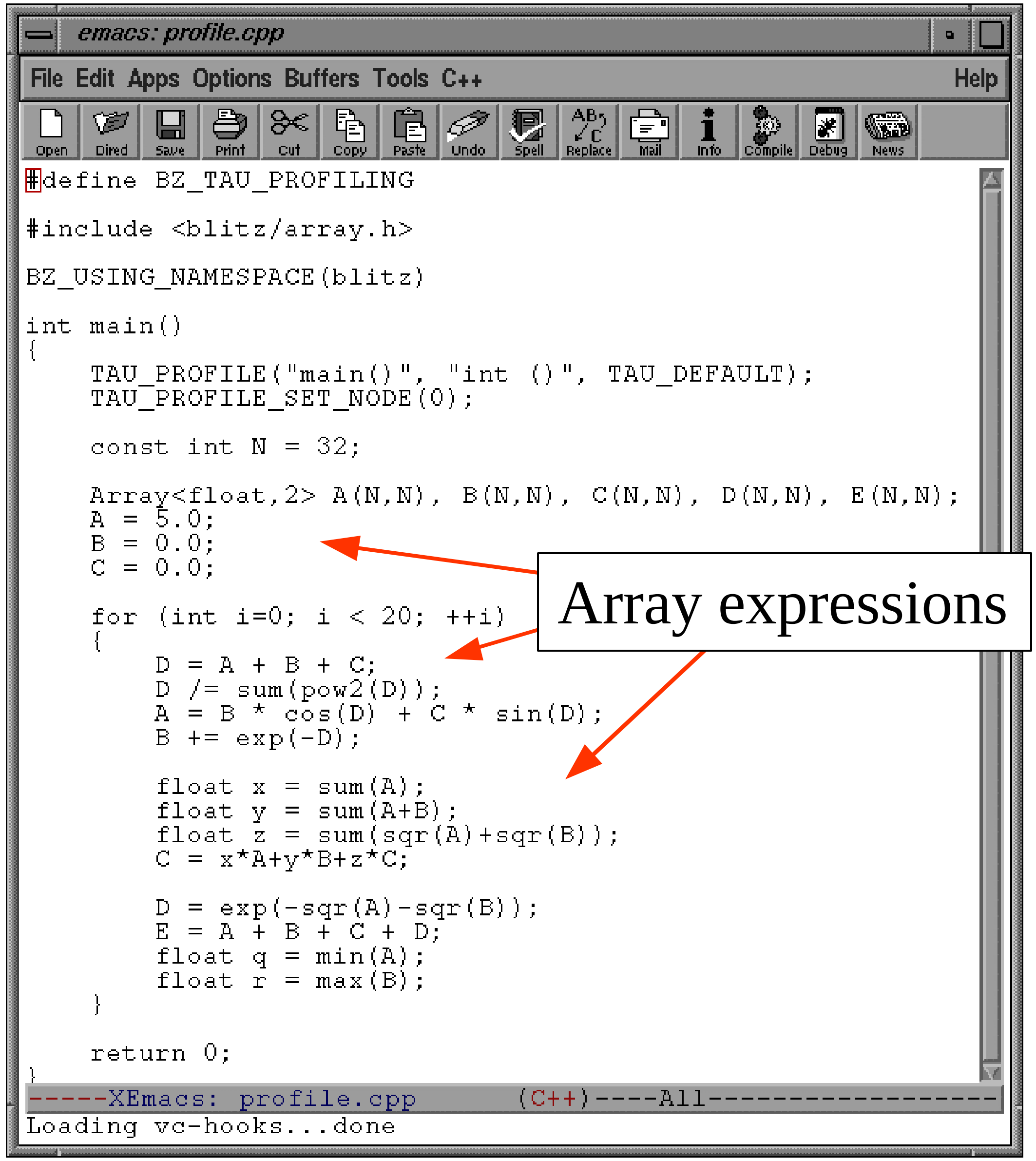
```
template<class T, unsigned Dim, class M, class C>
void Field<T,Dim,M,C>::initialize(
          Mesh_t&  m,
          FieldLayout<Dim>& l, const Bconds<T,Dim,M,C>& bc,
                                    const GuardCellSizes<Dim>& gc)
 {
   TAU_TYPE_STRING(taustr, “void (Mesh_t,” +CT(l) + “, “ +
                            CT(bc)+ “, “ + CT(gc) + “ )” );
   TAU_PROFILE(“Field::initialize()”, taustr, TAU_USER);
   BareField<T,Dim>::initialize(l,gc);
   store_mesh(&m, false);
  }
```

Multi-threaded Instrumentation Using TAU API

```
void *threaded_func (void *data)
{
  TAU_REGISTER_THREAD();
  TAU_PROFILE("threaded_func()", "void * (void *)",
                    TAU_DEFAULT);
  // do work here …
}
int main()
{
 TAU_PROFILE("main()", "int ()", TAU_DEFAULT);
  ret = pthread_create(&tid, NULL, threaded_func, NULL);
  // …
}
```

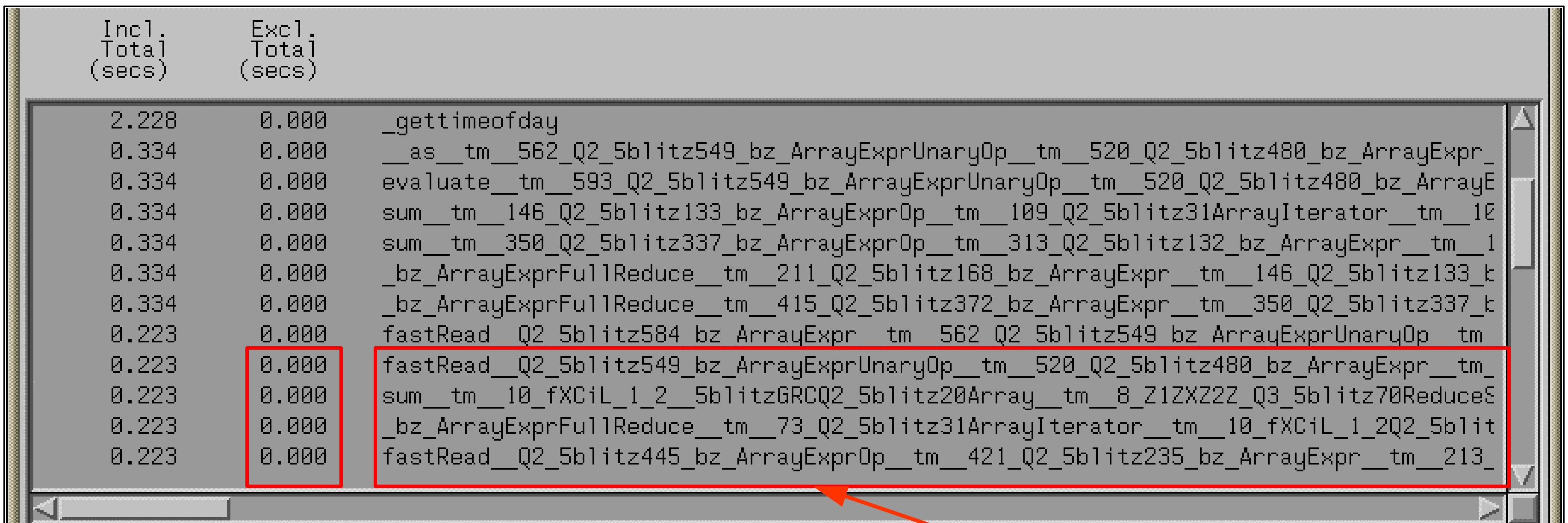
C++ Template Instrumentation (Blitz++, PETE)

- High-level objects
  - Array classes
  - Templates (Blitz++)
- Optimizations
  - Array processing
  - Expressions (PETE)
- Relate performance data to high-level statement
- Complexity of template evaluation



Standard Template Instrumentation Difficulties

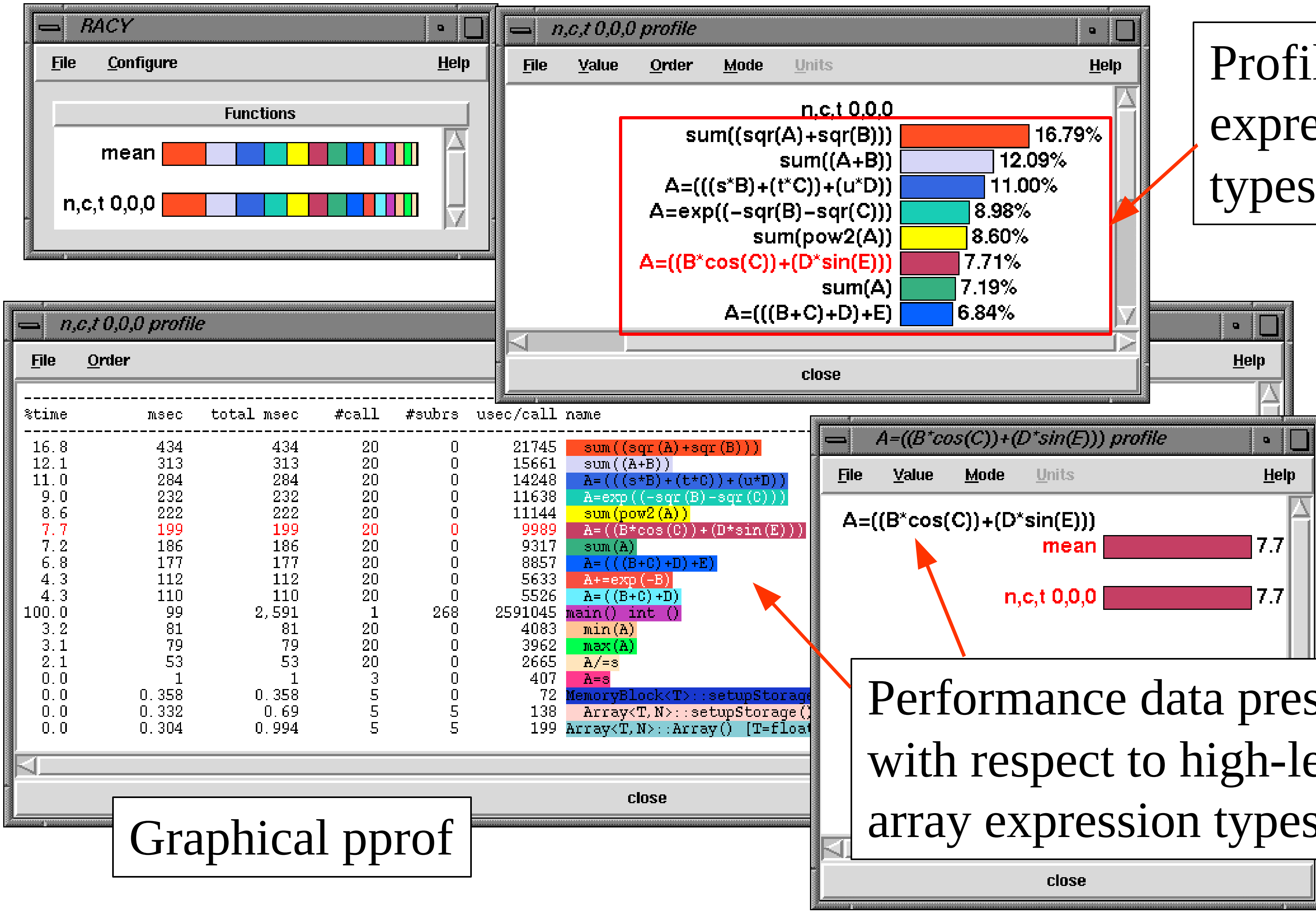
- Instantiated templates result in mangled identifiers
- Standard profiling techniques / tools are deficient
  - Integrated with proprietary compilers
  - Specific systems platforms and programming models



Incl. Total (secs)	Excl. Total (secs)	
2.228	0.000	_gettimeofday
0.334	0.000	__as__tm__562_Q2_5blitz549_bz_ArrayExprUnaryOp__tm__520_Q2_5blitz480_bz_ArrayExpr_
0.334	0.000	evaluate__tm__593_Q2_5blitz549_bz_ArrayExprUnaryOp__tm__520_Q2_5blitz480_bz_ArrayE
0.334	0.000	sum__tm__146_Q2_5blitz133_bz_ArrayExprOp__tm__109_Q2_5blitz31ArrayIterator__tm__1E
0.334	0.000	sum__tm__350_Q2_5blitz337_bz_ArrayExprOp__tm__313_Q2_5blitz132_bz_ArrayExpr__tm__1
0.334	0.000	_bz_ArrayExprFullReduce__tm__211_Q2_5blitz168_bz_ArrayExpr__tm__146_Q2_5blitz133_b
0.334	0.000	_bz_ArrayExprFullReduce__tm__415_Q2_5blitz372_bz_ArrayExpr__tm__350_Q2_5blitz337_b
0.223	0.000	fastRead__Q2_5blitz584_bz_ArrayExpr__tm__562_Q2_5blitz549_bz_ArrayExprUnaryOp__tm_
0.223	0.000	fastRead__Q2_5blitz549_bz_ArrayExprUnaryOp__tm__520_Q2_5blitz480_bz_ArrayExpr__tm_
0.223	0.000	sum__tm__10_fXCiL_1_2__5blitzGRCQ2_5blitz20Array__tm__8_Z1ZXZ2Z_Q3_5blitz70ReduceS
0.223	0.000	_bz_ArrayExprFullReduce__tm__73_Q2_5blitz31ArrayIterator__tm__10_fXCiL_1_2Q2_5blit
0.223	0.000	fastRead__Q2_5blitz445_bz_ArrayExprOp__tm__421_Q2_5blitz235_bz_ArrayExpr__tm__213_

Uninterpretable routine names

TAU Instrumentation and Profiling



Profile of expression types

Performance data presented with respect to high-level array expression types

Graphical pprof

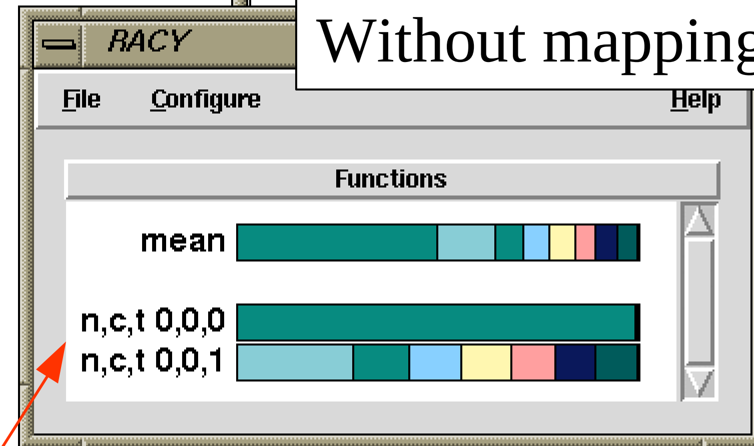
TAU and SMARTS: Asynchronous Performance

- Scalable Multithreaded Asynchronuous RTS
  - User-level threads, light-weight virtual processors
  - Macro-dataflow, asynchronous execution interleaving iterates from data-parallel statements
  - Integrated with POOMA II (parallel dense array library)
- Measurement of asynchronous parallel execution
  - Utilized the TAU mapping API
  - Associate iterate performance with data parallel statement
  - Evaluate different scheduling policies
- *"SMARTS: Exploting Temporal Locality and Parallelism through Vertical Execution" (ICS '99)*

TAU Mapping of Asynchronous Execution

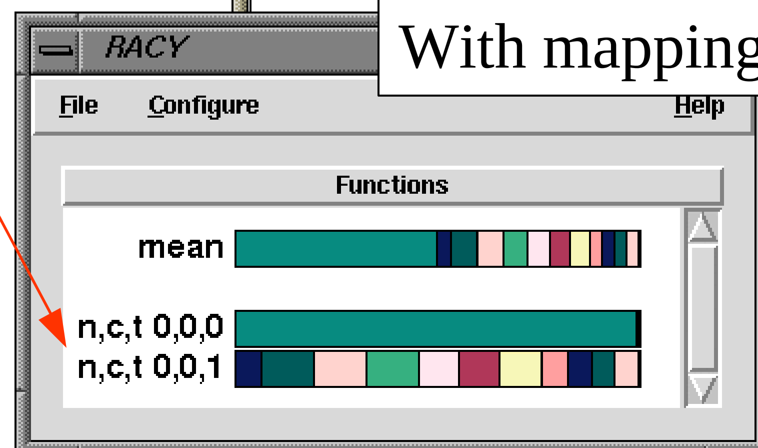
```
#include "Pooma/Arrays.h"

#include <iostream.h>

// The size of each side of the domain.
const int N = 3*1024;

int
main(
    int                  argc,          // argument count
    char *               argv[]         // argument list
){
    // Initialize Pooma.
    Pooma::initialize(argc, argv);

    // The array we'll be solving for
    Array<2> A(N, N), B(N,N), C(N,N), D(N,N), E(N,N);

    // Must block since we're doing some scalar code (see Tutorial 4).
    Pooma::blockAndEvaluate();

    A = 1.0;
    B = 2.0;
    C = 3.0;
    D = 4.0;
    E = 5.0;

    A = B + C + D;
    C = E - A + 2.0 * B;
    D = A + C;
    C = D + A - B;
    A = 2.0 * D + E ;
    E = 1.5 * B - A ;

    Pooma::blockAndEvaluate();

    cout << "D(1,1) = " << D(1,1) << endl;
    cout << "D(9,9) = " << D(9,9) << endl;

    // Clean up Pooma and report success.
    Pooma::finalize();
    return 0;
}
```

Without mapping



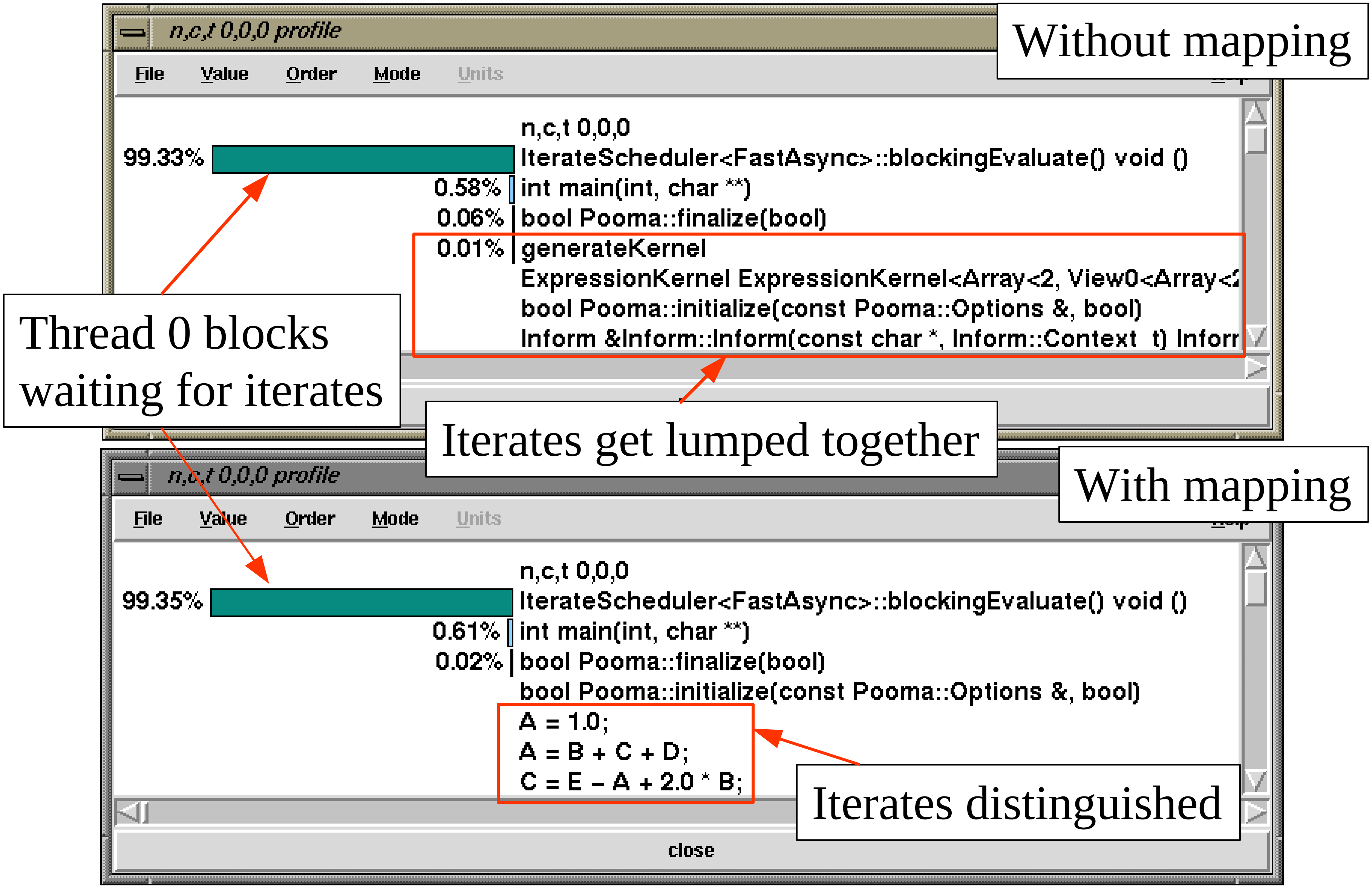
Two threads executing

With mapping

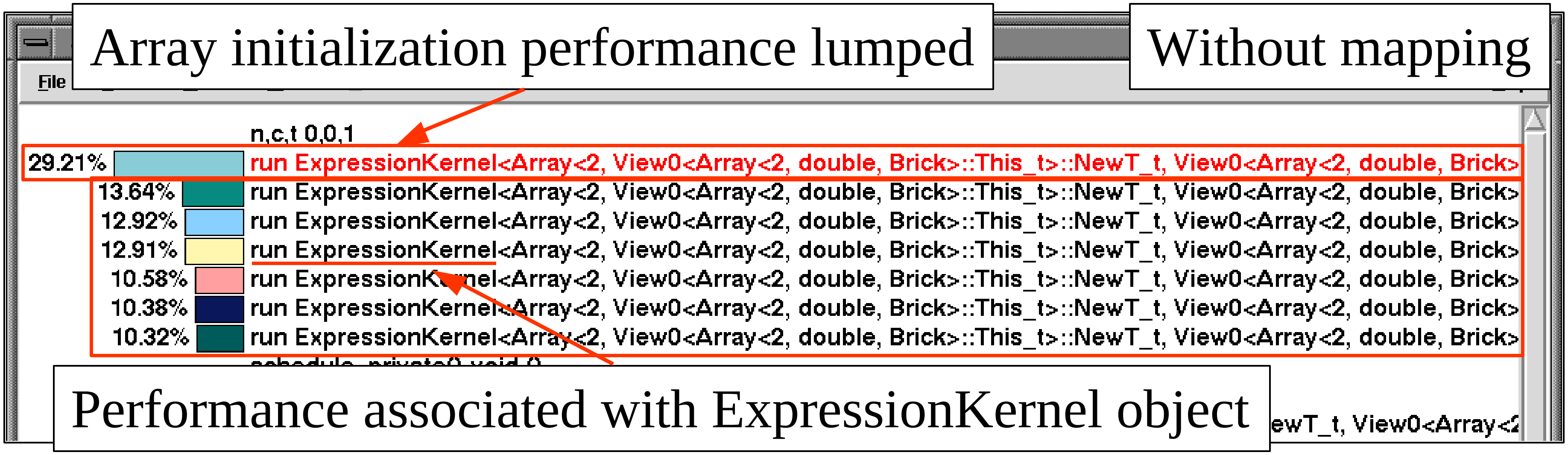


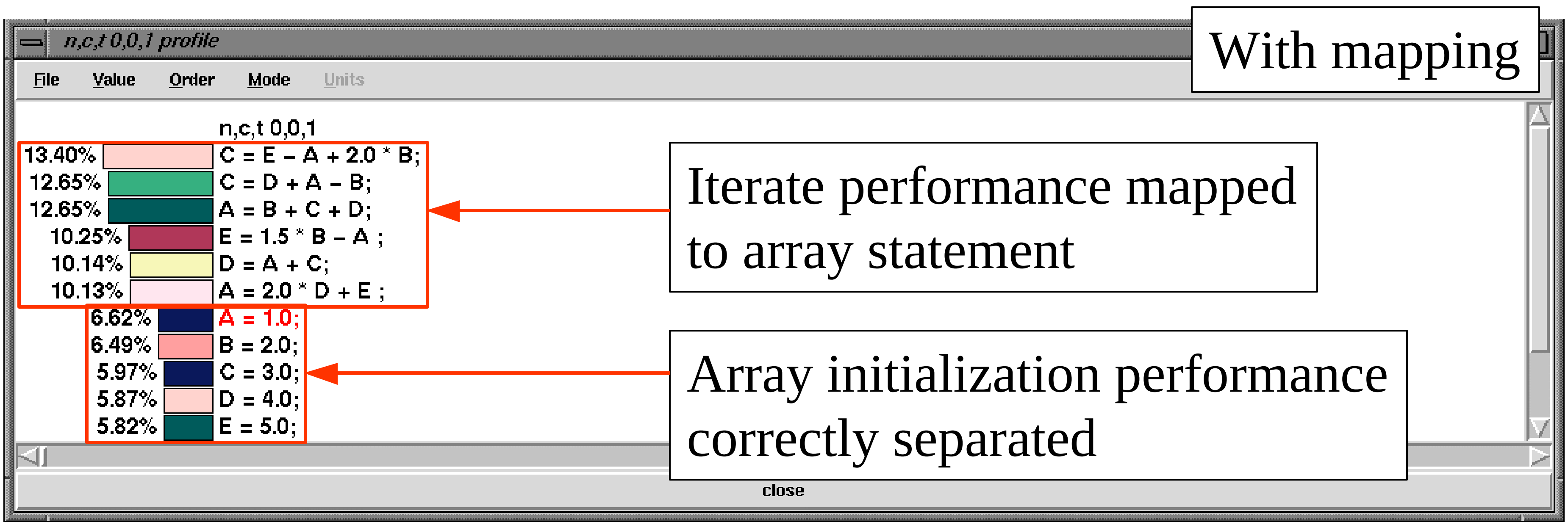
POOMA / SMARTS

With and Without Mapping (Thread 0)

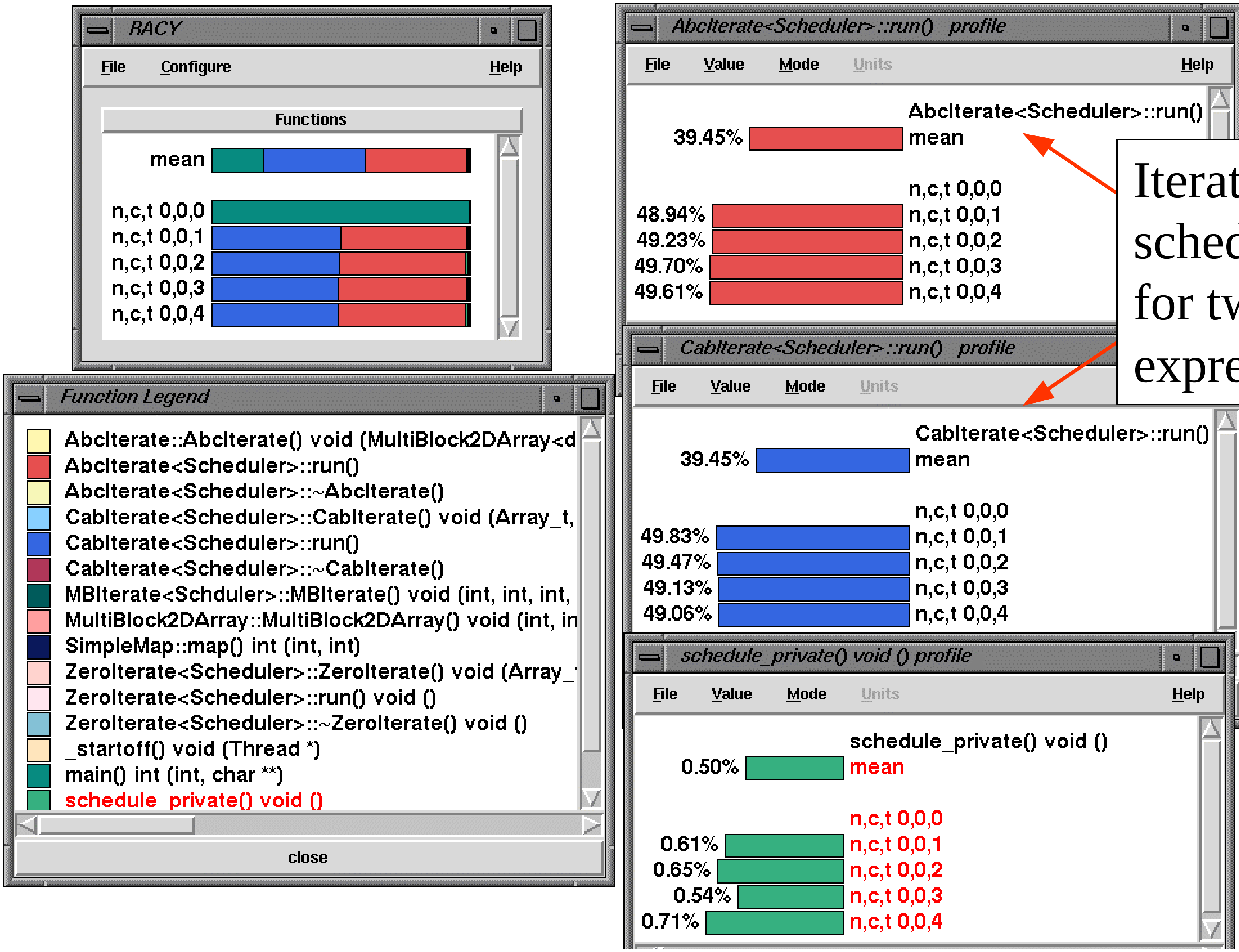


With and Without Mapping (Thread 1)





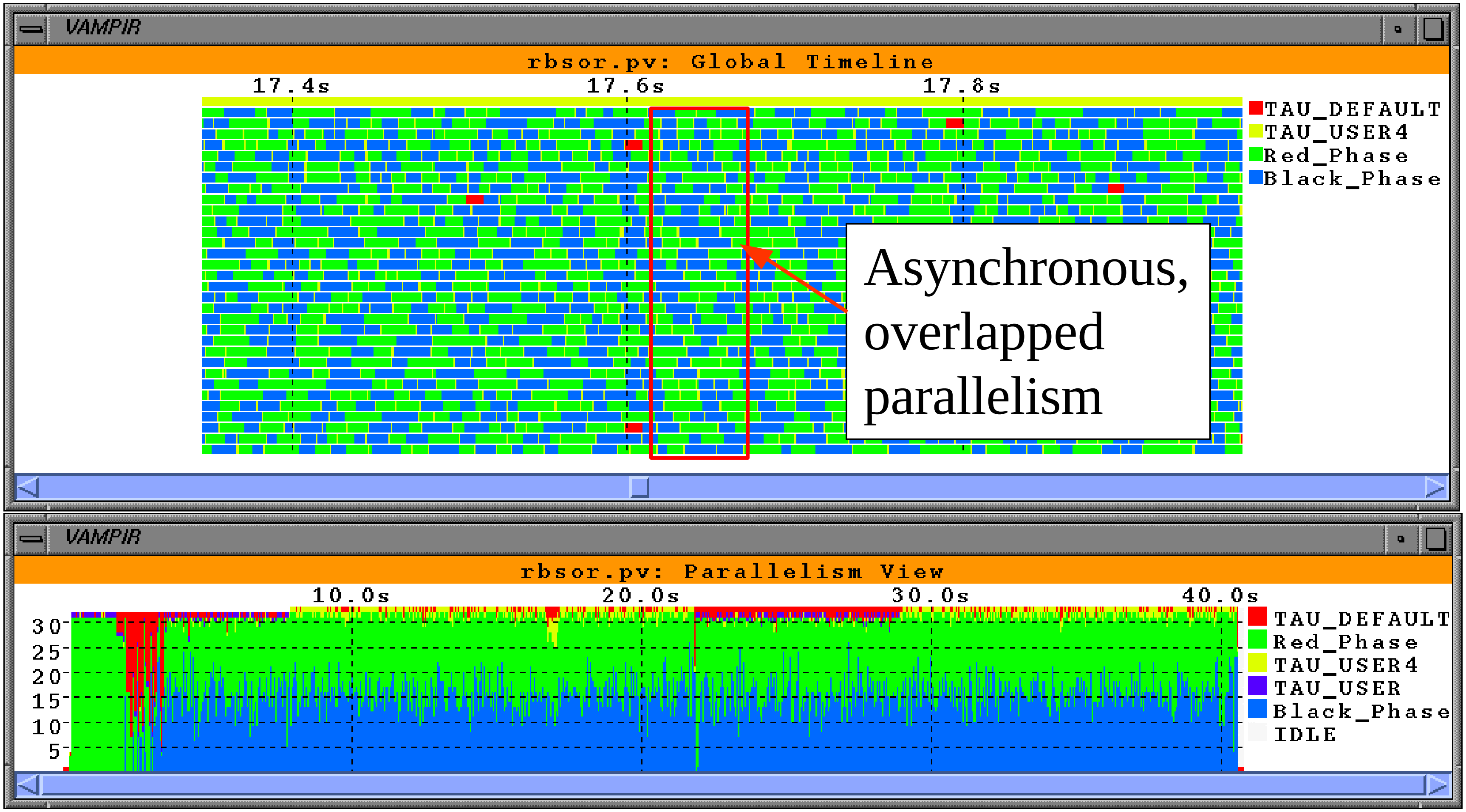
TAU Profiling of SMARTS Scheduling



Iteration scheduling for two array expressions

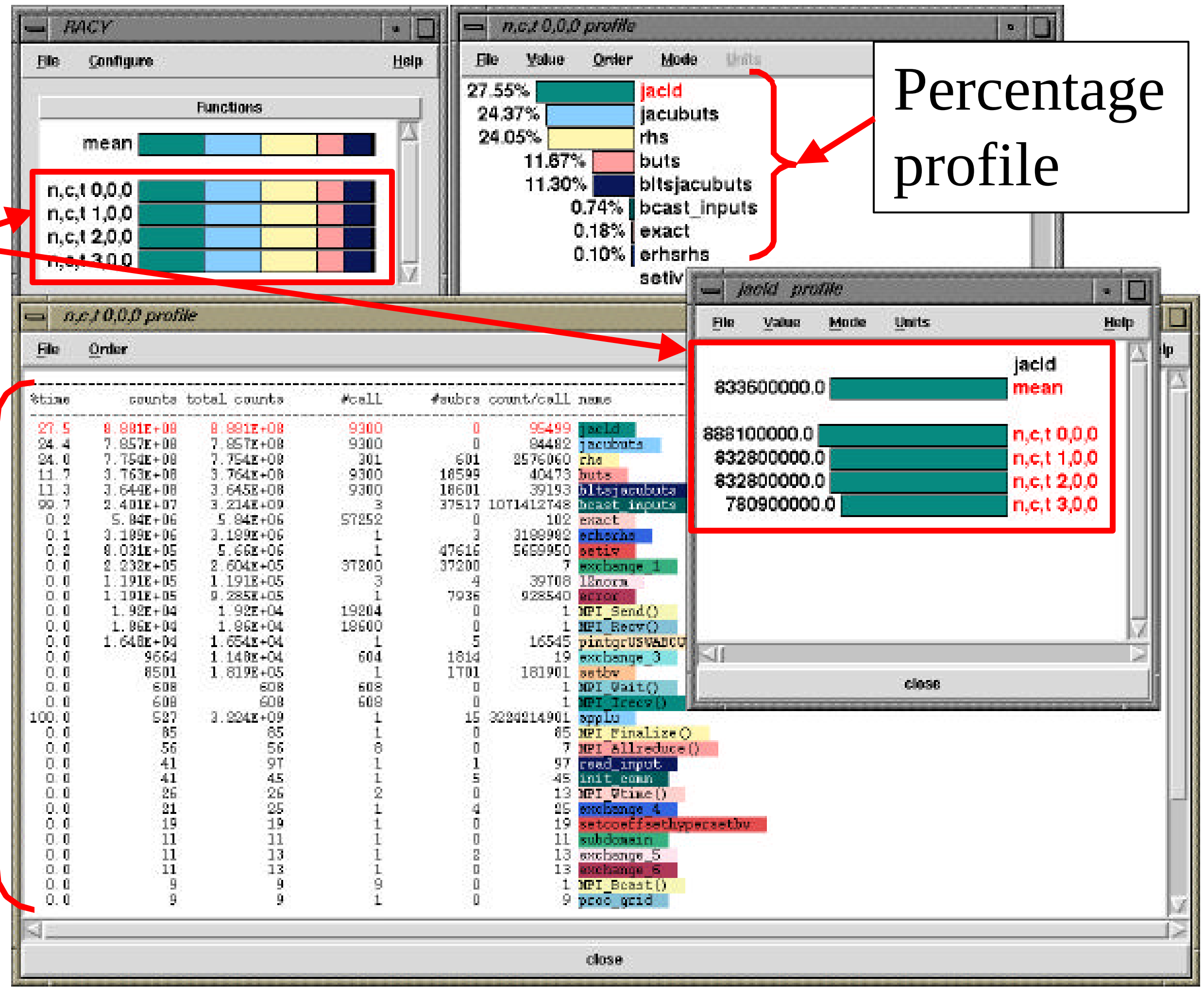
SMARTS Tracing (SOR) – Vampir Visualization

- SCVE scheduler used in Red/Black SOR running on 32 processors of SGI Origin 2000



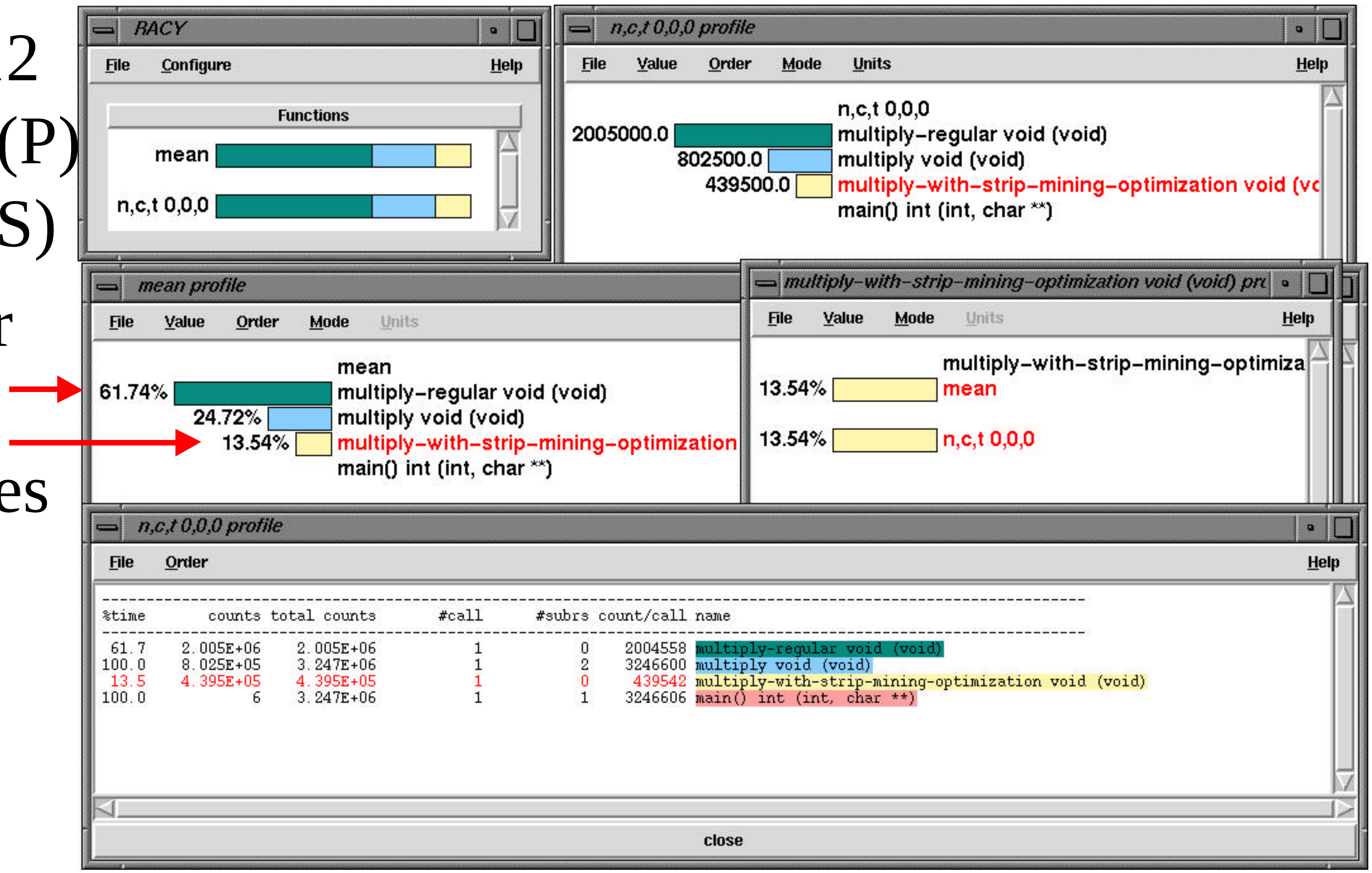
TAU and PAPI (NAS Parallel LU)

- SGI Power Onyx (4 processors, R10K), MPI
- Floating point operations
- Cross-node full / routine profiles
- Full FP profile for each node
- Counts in place of time



TAU and PAPI (Matrix Multiply)

- Data cache miss comparison,
- "regular" vs. "strip-mining" execution
- 512x512
  32 KB (P)
  2 MB (S)
- Regular causes 4.5 times more misses



Virtual Machine Execution (Java)

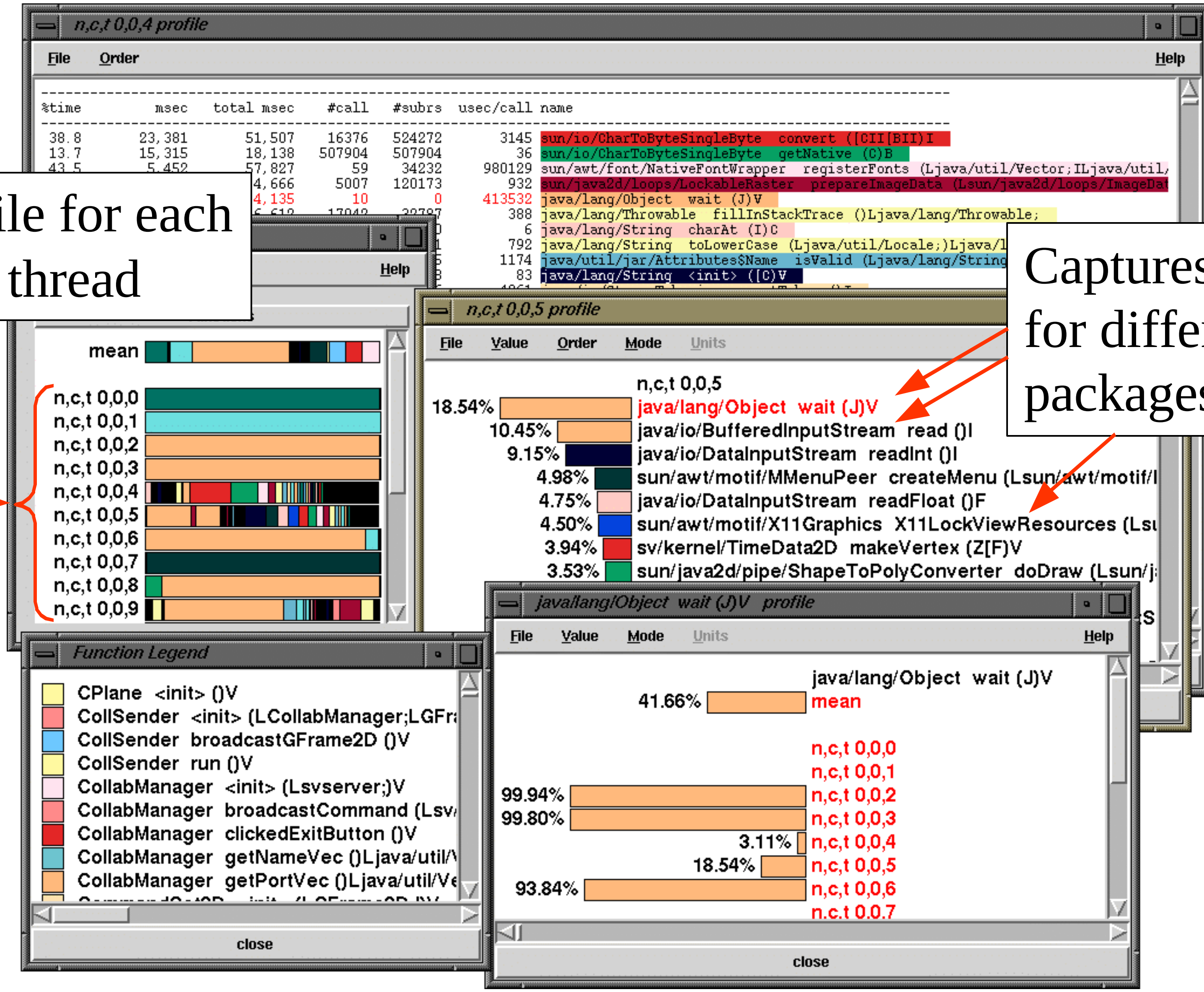
- Profile and trace Java (JDK 1.2+) applications
- No need to modify Java source, bytecode, or JVM
- Implemented using JVMPI (JVM profiling interface)
  - Fields JVMPI events
- Executes in memory space of JVM
  - Profiler agent loaded as shared object
- Usage (SciVis, NPAC, Syracuse University)

```
% ./configure -jdk=<dir_where_jdk_is_installed>
% setenv LD_LIBRARY_PATH
  $LD_LIBRARY_PATH\:<taudir>/<arch>/lib
% java -XrunTAU svserver
```

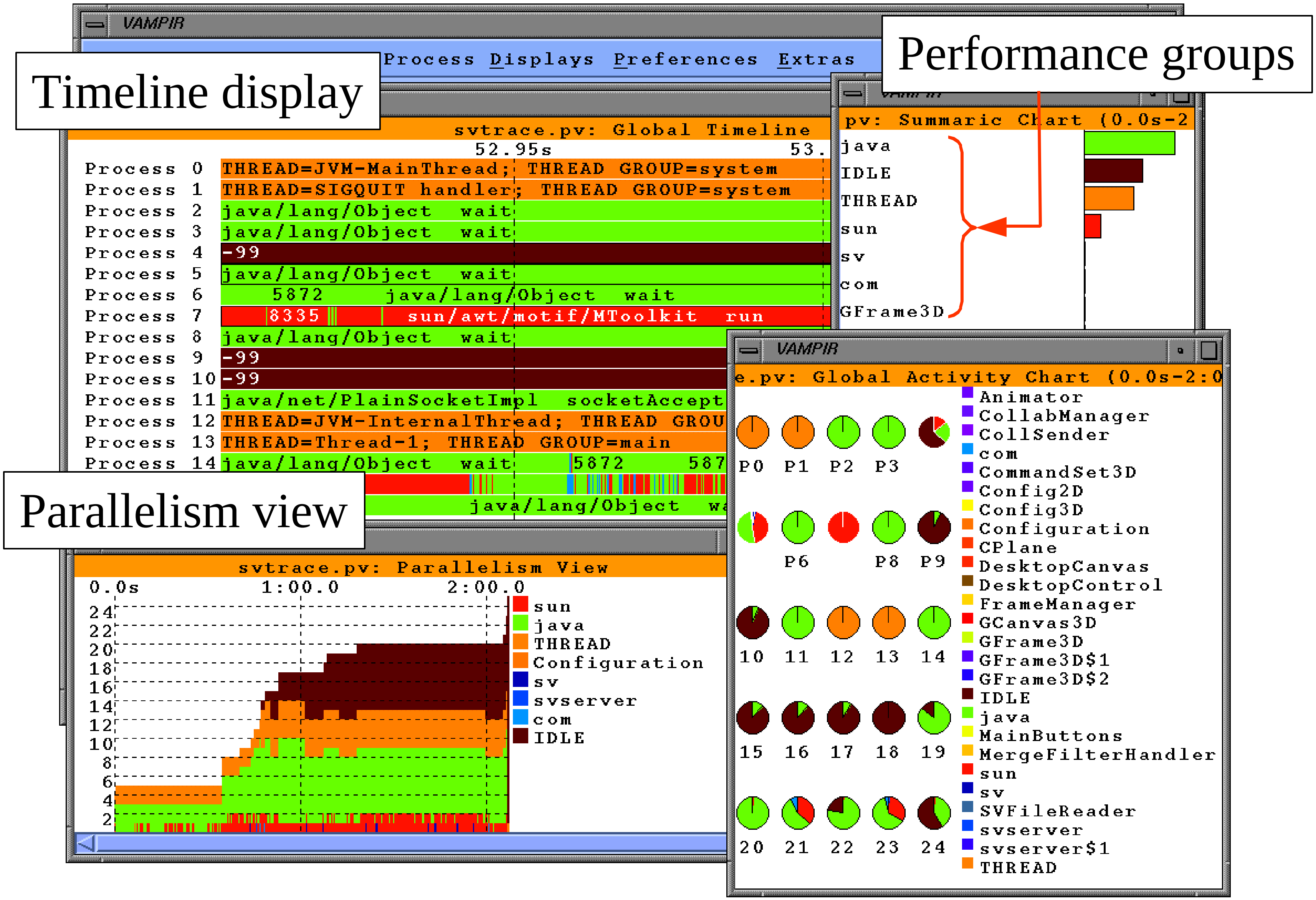
TAU Profiling of Java Application (SciVis)

Profile for each Java thread

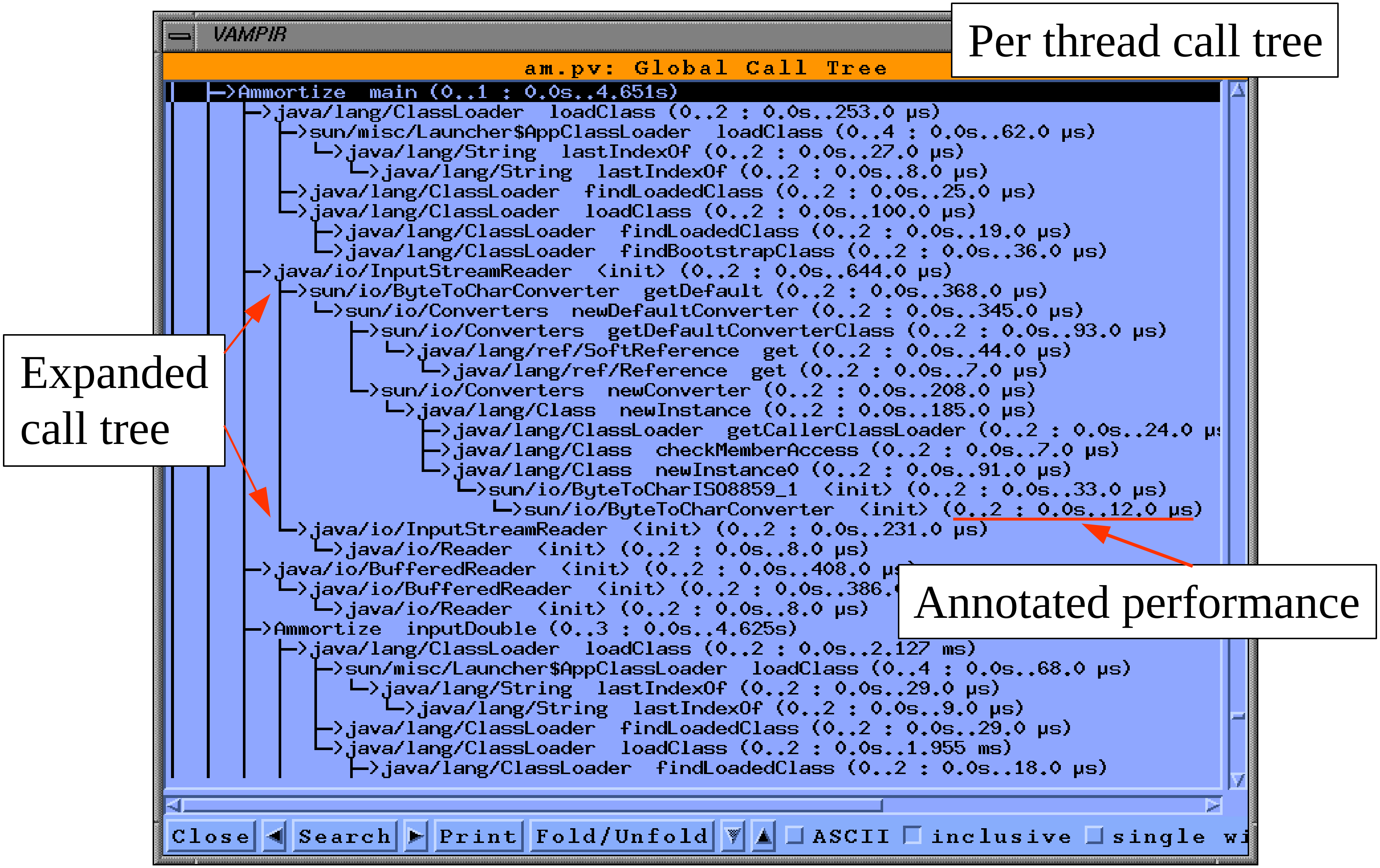
Captures events for different Java packages



Java Tracing (SciVis) – Vampir Visualization



Vampir Dynamic Call Tree View (SciVis)



Using TAU with JAVA

- For Profiling

  % configure -jdk=/usr/local/packages/jdk

  % make clean; make install

  % setenv LD_LIBRARY_PATH
  $LD_LIBRARY_PATH\:/usr/tau-2.x/solaris2/lib

  % java -XrunTAU java_app

  % racy

- For Tracing

  % configure -jdk=/usr/local/packages/jdk -TRACE

  % make clean; make install

  % java -XrunTAU java_app

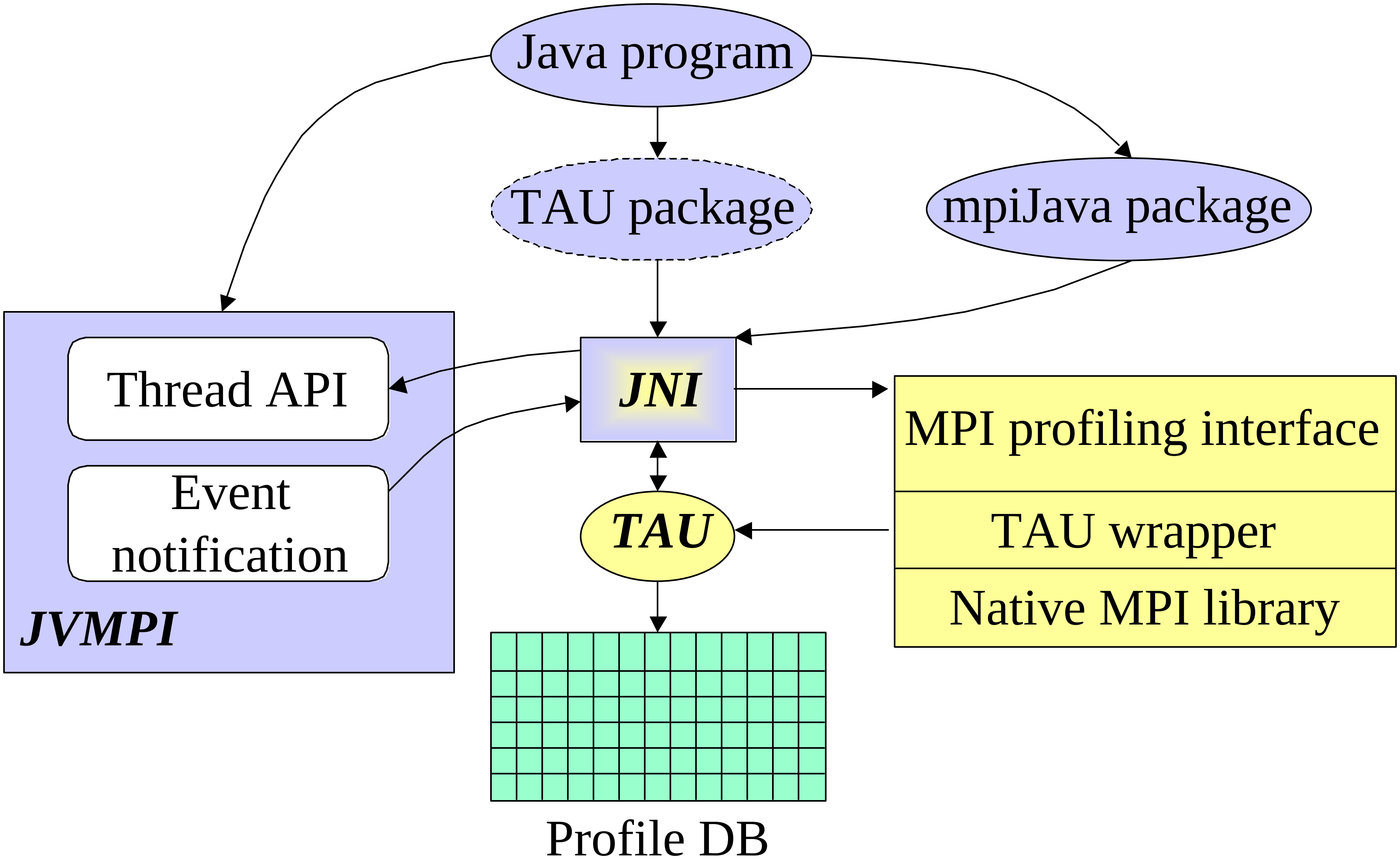
  % tau_merge *.trc app.trc; tau_convert -vampir app.trc tau.edf app.pv

  % vampir app.pv

Hybrid Parallel Computation (Java + MPI)

- Multi-language applications and hybrid execution
  - Java, C, C++, Fortran
  - Java threads and MPI
- mpiJava (Syracuse, JavaGrande)
  - Java wrapper package with JNI C bindings to MPI routines
- Integrate cross-language/system performance technology
  - JVMPI and Tau profiler agent
  - MPI profiling interface - link-time interposition library
  - Cross execution mode uniformity and consistency
    - invoke JVMPI control routines to control Java threads
    - access thread information and expose to MPI interface
- *"Performance Tools for Parallel Java Environments," Java Workshop, ICS 2000, May 2000.*

TAU Java Instrumentation Architecture



Parallel Java Game of Life (Profile)

- mpiJava testcase
- 4 nodes, 28 threads

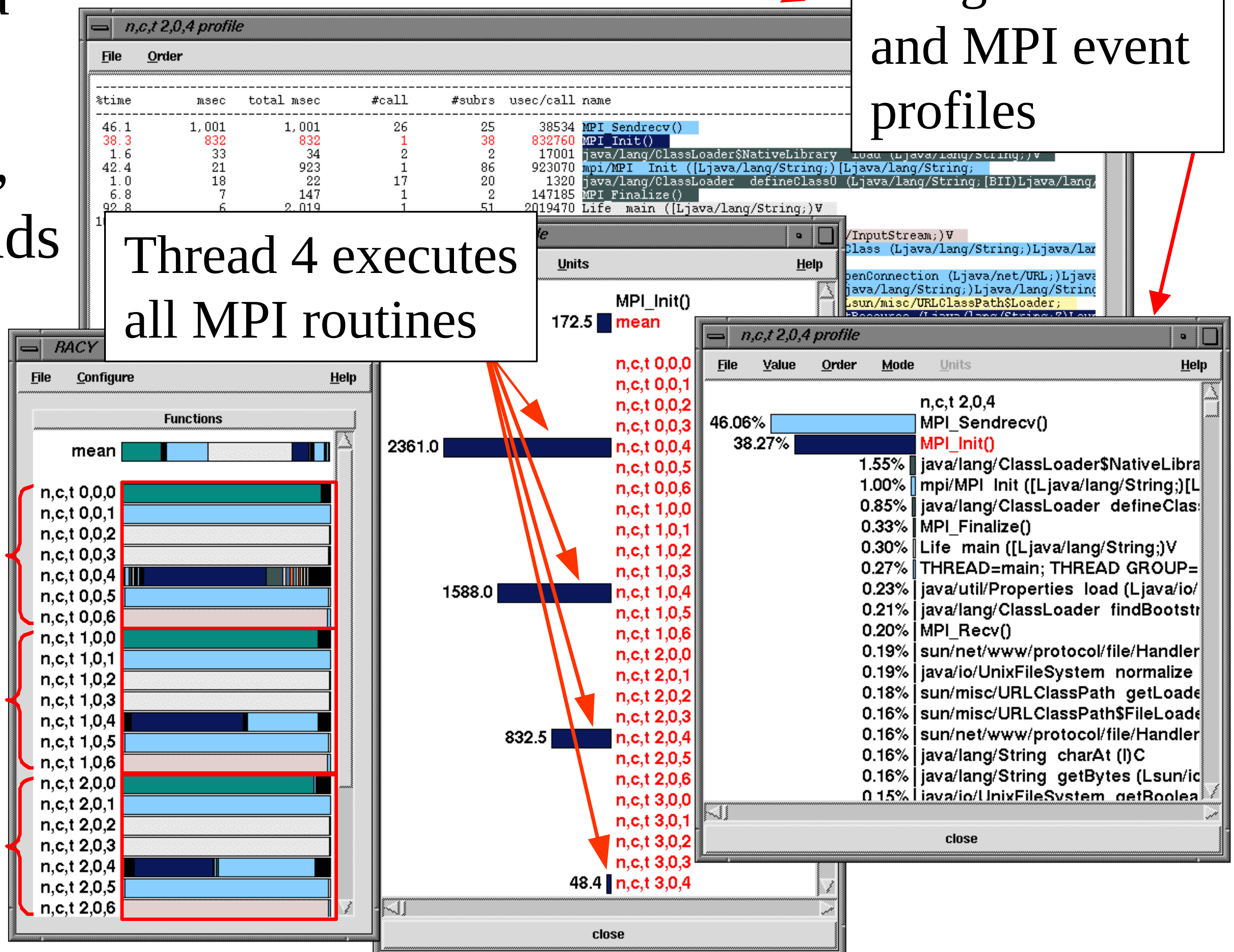
Merged Java and MPI event profiles

Thread 4 executes all MPI routines

Node 0

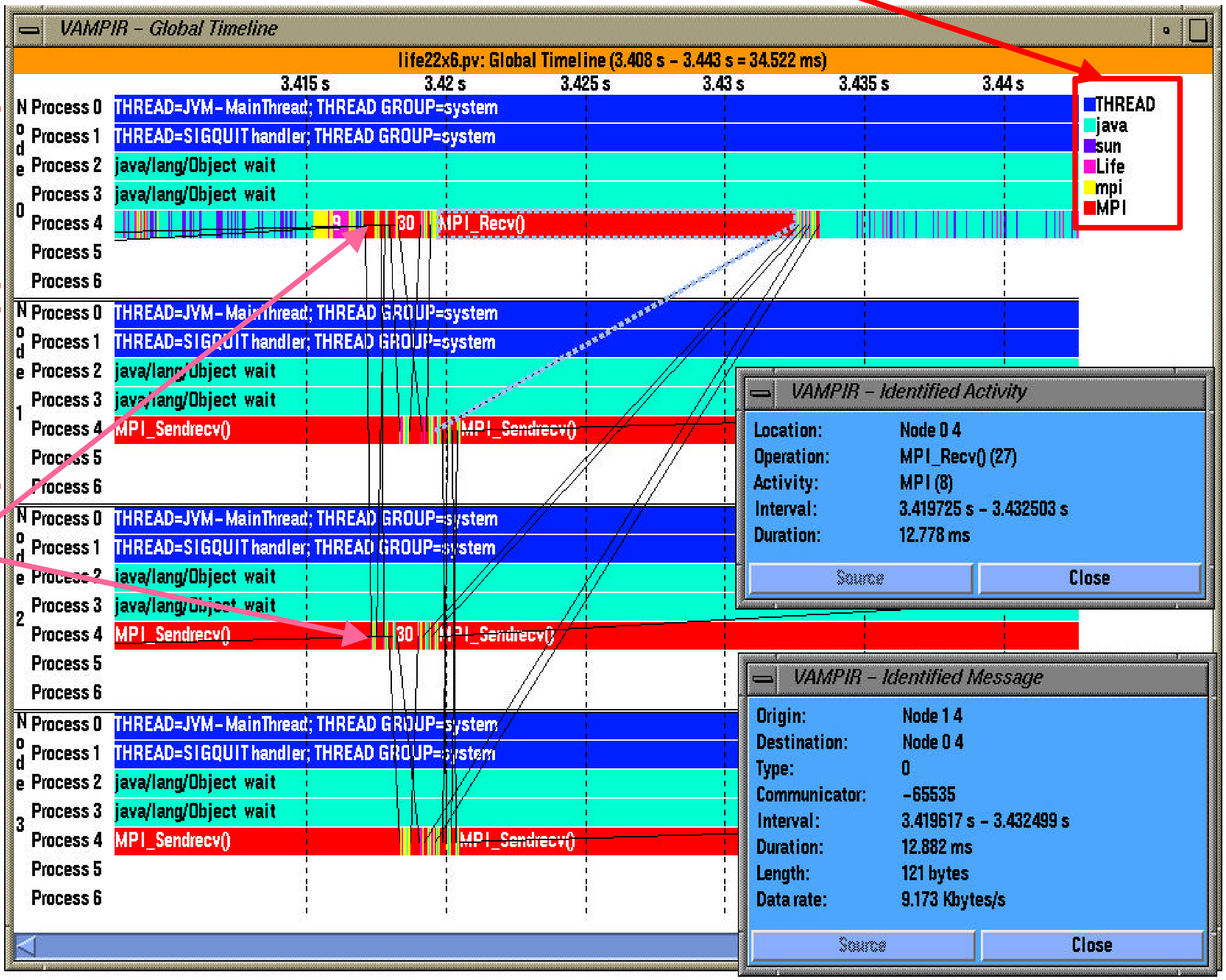
Node 1

Node 2



Parallel Java Game of Life (Trace)

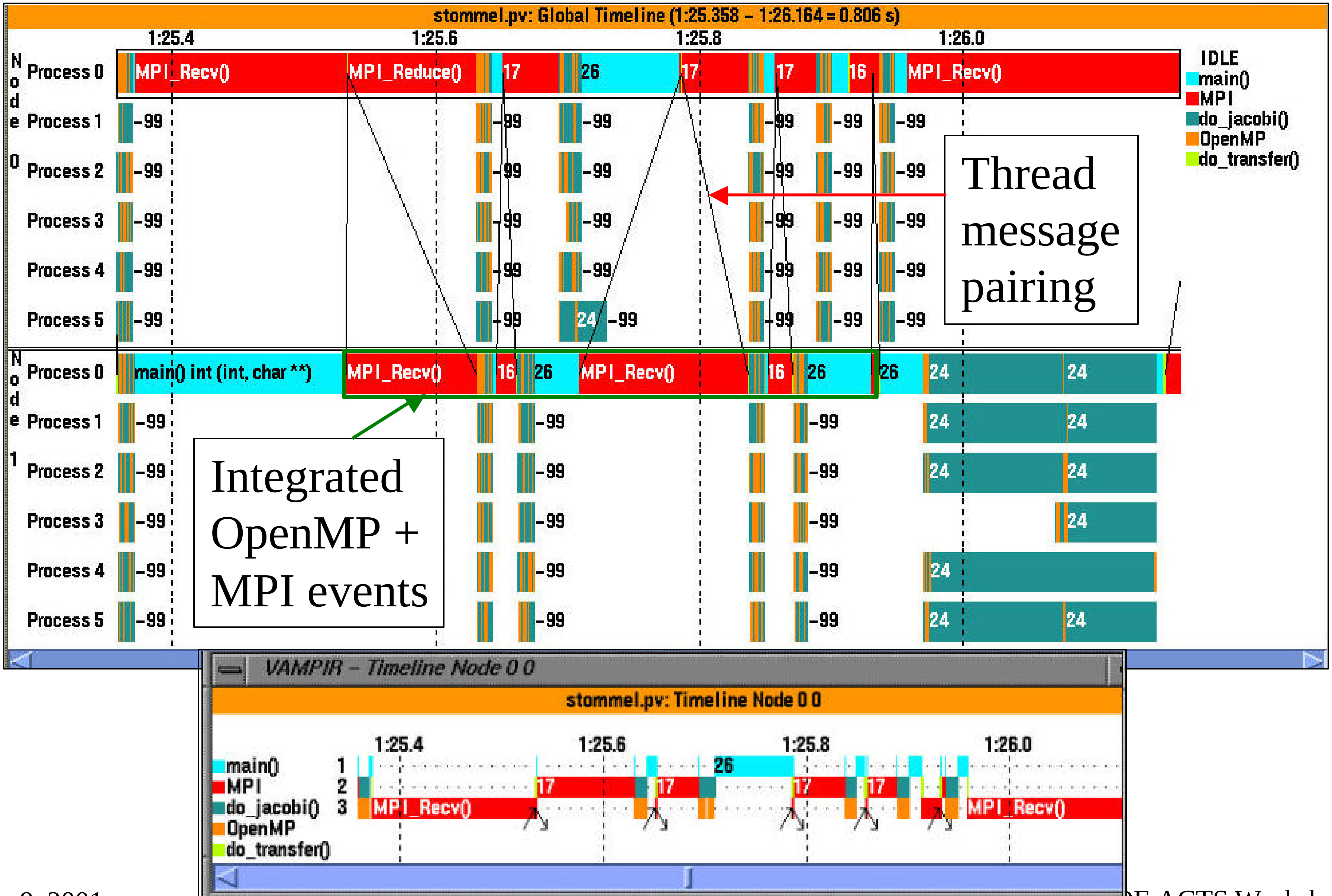
- Integrated event tracing
- Multi-level event grouping
- Merged trace viz
- Node process grouping
- Thread message pairing
- Vampir display



Hybrid Parallel Computation (OpenMP + MPI)

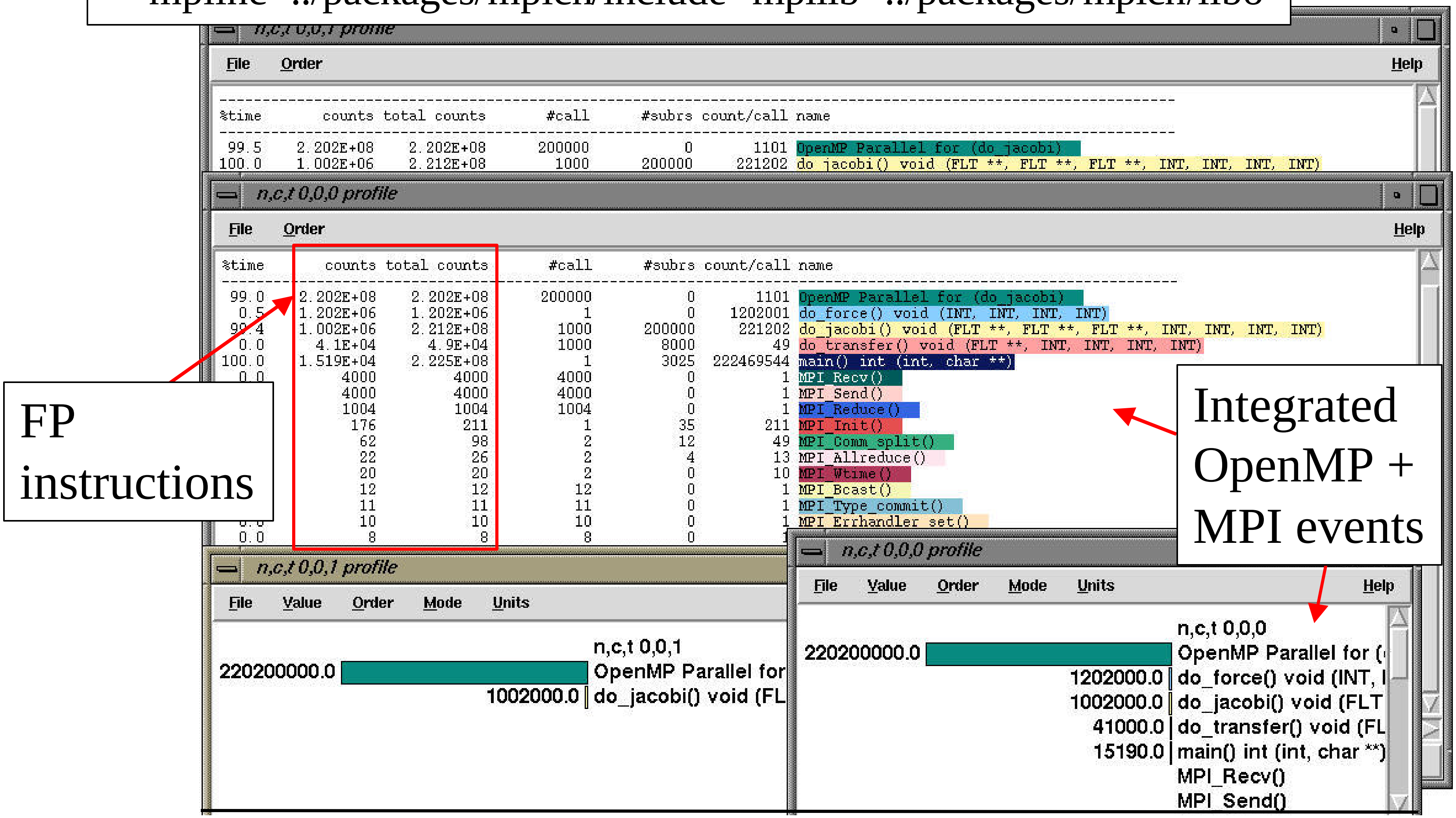
- Portable hybrid (mixed model) parallel programming
  - OpenMP for shared memory parallel programming
    - fork-join model
    - loop level parallelism
  - MPI for cross-box message-based parallelism
- OpenMP performance measurement
  - Interface to OpenMP runtime system (RTS events)
  - Compiler support and integration
- 2D Stommel model of ocean circulation
  - Jacobi iteration, 5-point stencil
  - Timothy Kaiser (San Diego Supercomputing Center)

OpenMP + MPI Ocean Modeling (Trace)



OpenMP + MPI Ocean Modeling (HW Profile)

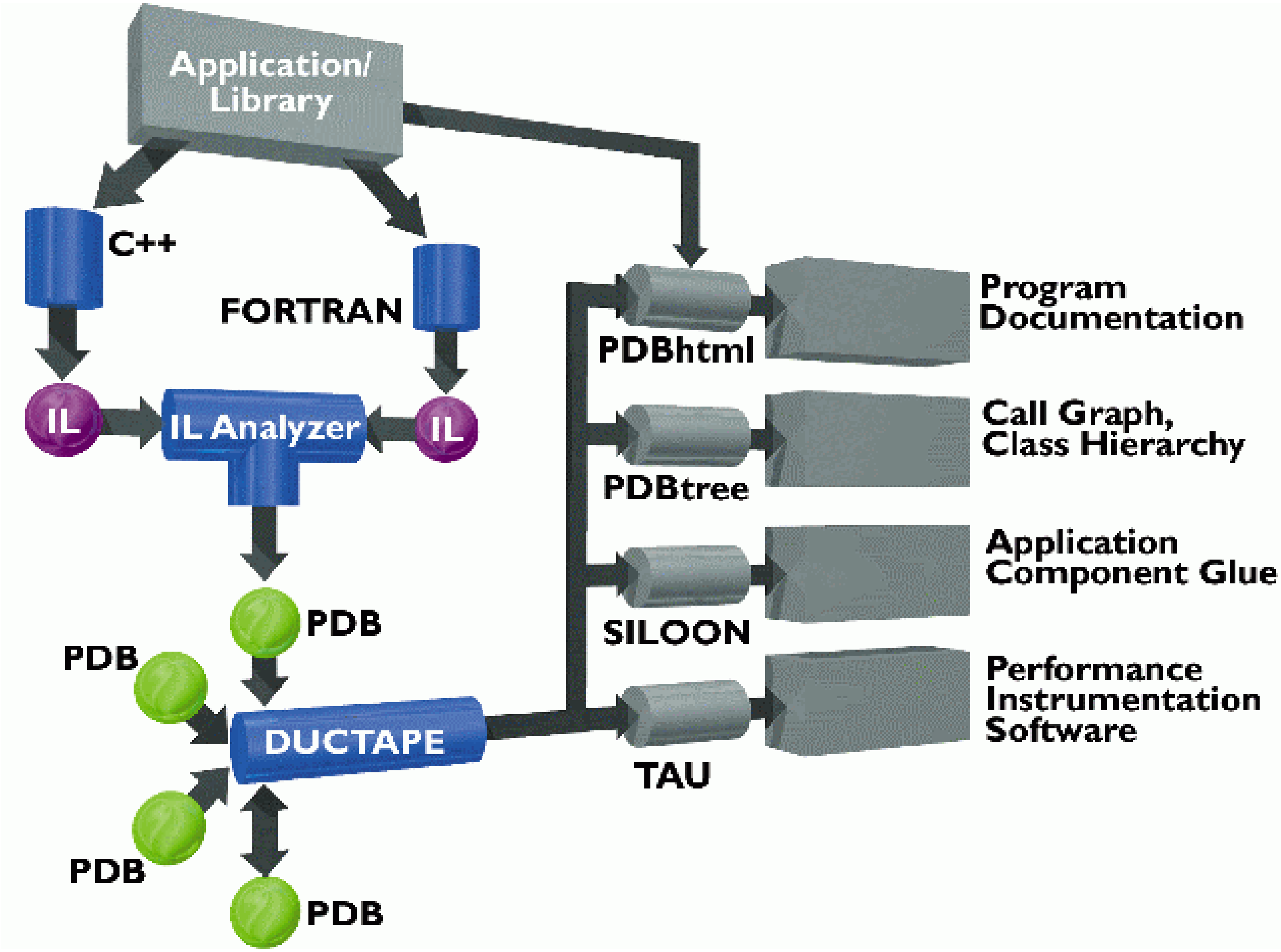
% configure -papi=../packages/papi -openmp -c++=pgCC -cc=pgcc
-mpiinc=../packages/mpich/include -mpilib=../packages/mpich/libo



Program Database Toolkit (PDT)

- Program code analysis framework for developing source-based tools
- High-level interface to source code information
- Integrated toolkit for source code parsing, database creation, and database query
  - commercial grade front end parsers
  - portable IL analyzer, database format, and access API
  - open software approach for tool development
- Target and integrate multiple source languages
- *http://www.acl.lanl.gov/pdtoolkit*

PDT Architecture and Tools



PDT Components

- Language front end
  - parses a C, C++, F77/F90 (soon), Java (next year)
    - Edison Design Group (EDG): C, C++, Java
    - Mutek Solutions Ltd.: F77, F90
    - academic license allows derivative tool distribution
  - creates an intermediate-language (IL) tree
- IL Analyzer
  - processes the intermediate language (IL) tree
  - creates “program database” (PDB) formatted file
    - more easily read by program or scripting language

PDT Components (continued)

- DUCTAPE (Bernd Mohr, ZAM, Germany)
  - C++ program Database Utilities and Conversion Tools APplication Environment
  - processes and merges PDB files
  - C++ library to access the PDB for PDT applications
- Sample Applications
  - *pdbmerge* : merges PDB files from separate analyses
  - *pdbconv* : converts PDB files to more readable format
  - *pdbtree* : prints file inclusion, class hierarchy, and call graph information
  - *pdbhtml* : ”HTMLizes" C++ source

PDT and TAU Instrumentation

- Manual source instrumentation
  - time consuming and error prone
- Automatic source instrumentation
  - need function and method signature
  - need parameter type information
  - need source file and line information
  - generate instrumentation statement
  - insert instrumentation in source file
- Use PDT to create/access program code information
- Develop instrumentation tool

PDT Summary

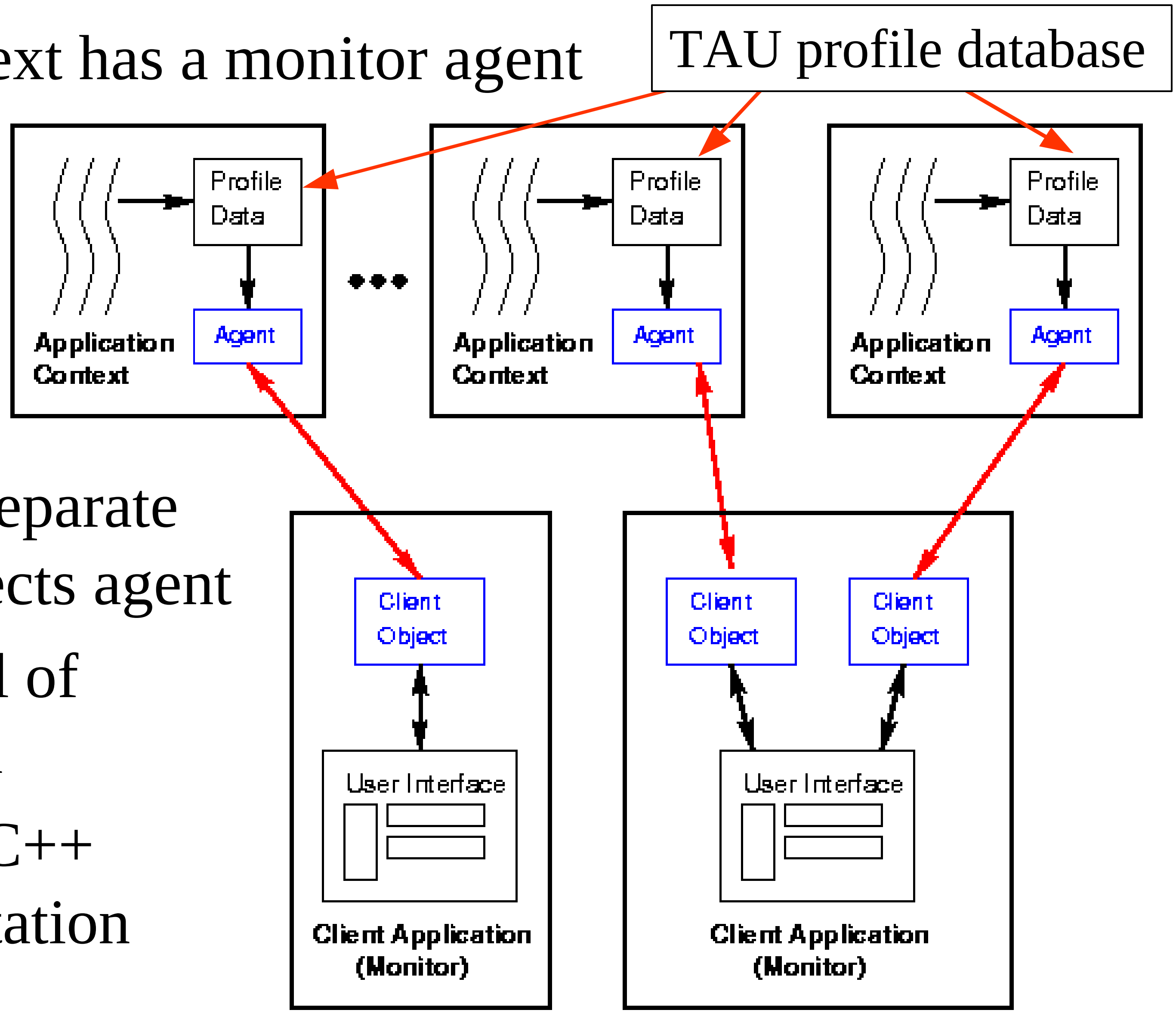
- Program Database Toolkit (Version 1.2)
  - EDG C++ Front End (Version 2.41.2)
  - C++ IL Analyzer and DUCTAPE library
  - tools: *pdbmerge*, *pdbconv*, *pdbtree*, *pdbhtml*
  - standard C++ system header files (KAI KCC 3.4c)
- Fortran 90 IL Analyzer in progress
- Automated TAU performance instrumentation
- Program analysis support for SILOON (ACL CD)
- *"A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software" (SC 2000)*

TAU Distributed Monitoring Framework

- Extend usability of TAU performance analysis
- Access TAU performance data during execution
- Framework model
  - each application context is a performance data server
  - monitor agent thread is created within each context
  - client processes attach to agents and request data
  - server thread synchronization for data consistency
  - pull mode of interaction
- Distributed TAU performance data space
- *“A Runtime Monitoring Framework for the TAU Profiling System” (ISCOPE ‘99)*

TAU Distributed Monitor Architecture

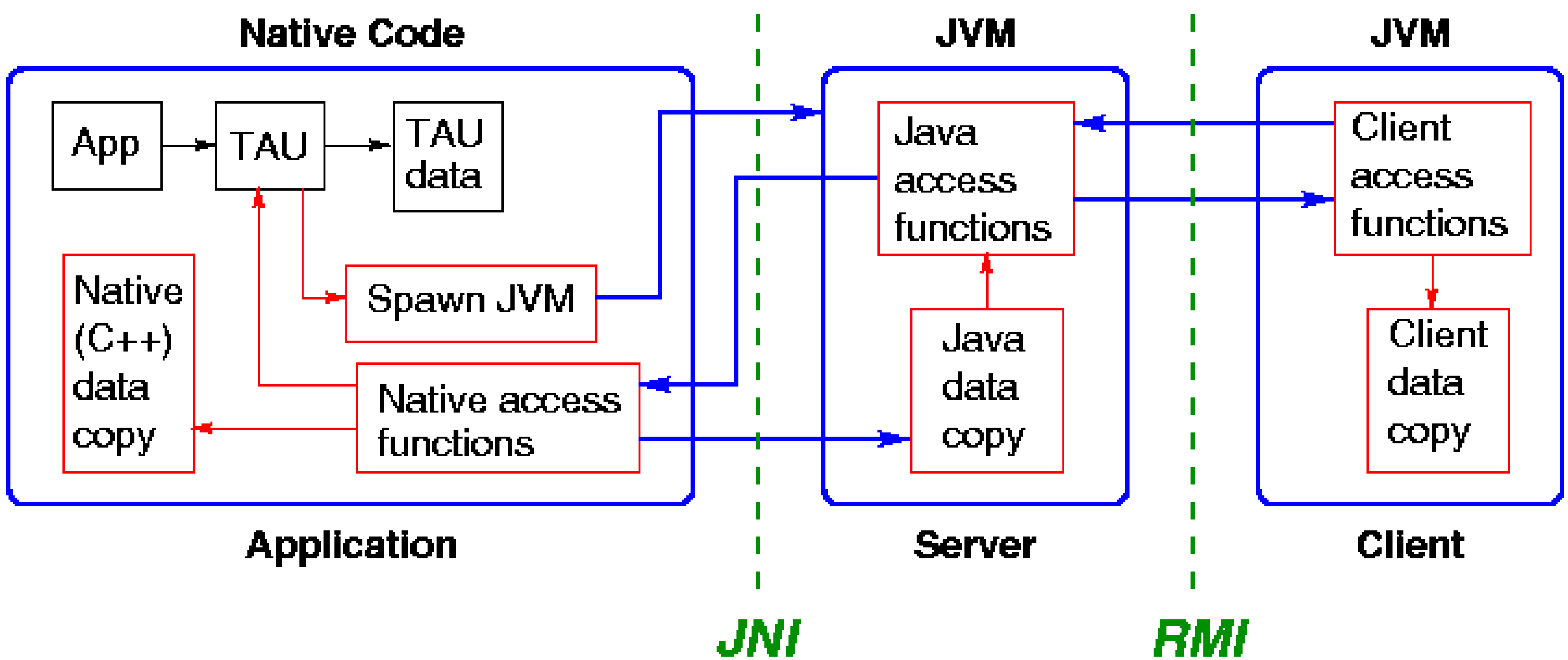
- Each context has a monitor agent



- Client in separate thread directs agent
- Pull model of interaction
- Initial HPC++ implementation

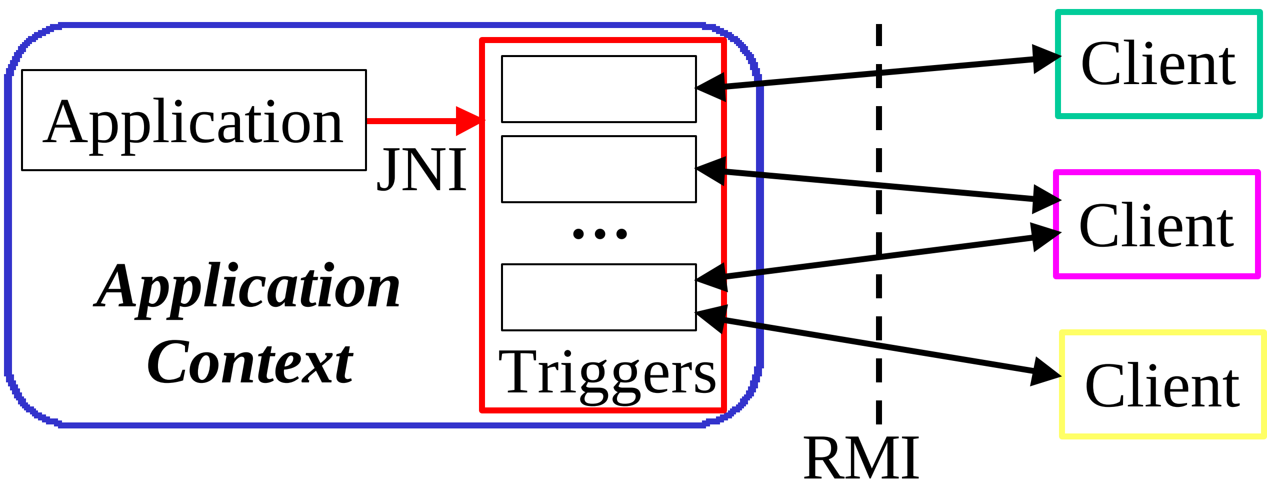
Java Implementation of TAU Monitor

- Motivations
  - More portable monitor middleware system (RMI)
  - More flexible and programmable server interface (JNI)
  - More robust client development (EJB, JDBC, Swing)



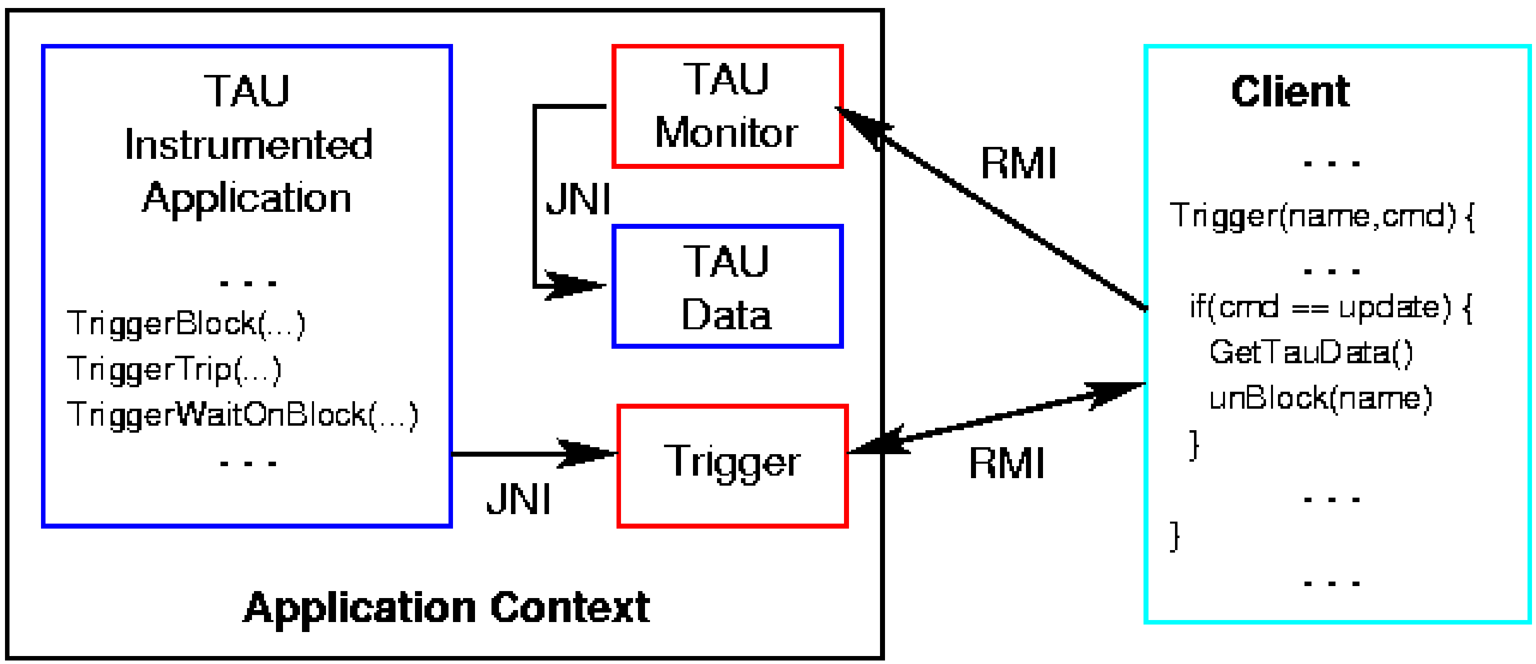
Trigger Support for Runtime Monitoring

- Execution event triggering
  - Inform external clients of events during execution
- "Server" library
  - Java trigger modules
  - JNI link between application and trigger modules
- "Client" trigger library



Trigger API and TAU Monitor Application

- Trigger at points of desired monitor access
- Pull TAU profile data
- Unblock trigger and continue



TAU Future Plans

- Platforms
  - IA-64, Compaq, Itanium, Sun Starfire, IBM Linux, ...
- Languages
  - OpenMP, Java (Java Grande), Opus / Java
- Instrumentation
  - Automatic (F90, Java), DynInst, DITools
- Measurement
  - Extend tracing support to include event data (e.g., HW counts)
  - Dynamic performance measurement control
- Displays
  - Extensible Performance Display Tool (ExPeDiTo)
  - TraceView 2 (TV2), Pajé
- Performance database and technology
  - Support for multiple runs
  - Open API for analysis tool development

Conclusions

- Complex parallel computing environments require robust and widely available performance technology
  - Portable, cross-platform, multi-level, integrated
  - Able to bridge and reuse existing technology
  - Technology savvy and open
- TAU is only a performance technology framework
  - General computation model and core services
  - Mapping, extension, and refinement
  - Integration of additional performance technology
- Need for higher-level framework layers
  - Computational and performance model archetypes
  - Performance diagnosis